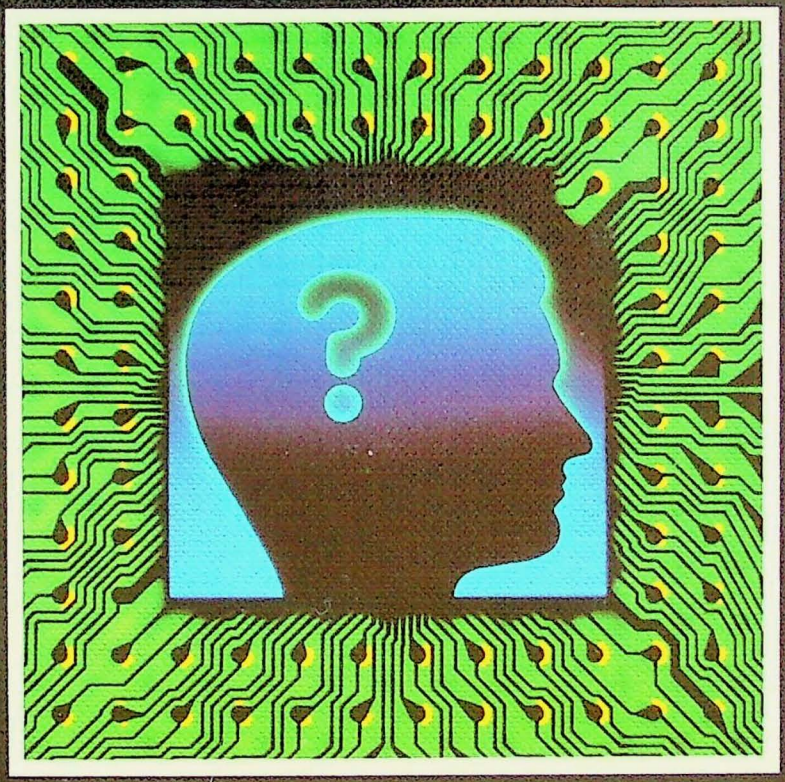
SYBEX

Einführung in Turbo Prolog



Carl Townsend

Einführung in Turbo Prolog

Carl Townsend

DÜSSELDORF · SAN FRANCISCO · PARIS · LONDON

Anmerkung:
Turbo Prolog ist ein geschütztes Warenzeichen von Borland Int. Inc., USA.

Originalausgabe in Englisch
Titel der amerikanischen Ausgabe: Introduction to Turbo Prolog
Original Copyright © SYBEX 1987 by SYBEX Inc., Berkley, USA

Deutsche Übersetzung: Gabriele Wentges

Umschlaggestaltung: Alfred Pasieka / tgr
Satz: B. Franke
Gesamtherstellung: Graf & Pflügge, Düsseldorf

Der Verlag hat alle Sorgfalt walten lassen, um vollständige und akkurate Informationen in diesem Buch bzw. Programm und anderen evtl. beiliegenden Informationsträgern zu publizieren. SYBEX-Verlag GmbH, Düsseldorf, übernimmt keine Garantie noch die juristische Verantwortung oder irgendeine Haftung für die Nutzung dieser Informationen, für deren Wirtschaftlichkeit oder fehlerfreie Funktion für einen bestimmten Zweck. Ferner kann der SYBEX-Verlag für Schäden, die auf eine Fehlfunktion von Programmen, Schaltplänen o.ä. zurückzuführen sind, nicht haftbar gemacht werden, auch nicht für die Verletzung von Patent- und anderen Rechten Dritter, die daraus resultieren.

ISBN-Nr. 3-88745-680-7
2. Auflage 1988

Alle Rechte vorbehalten. Kein Teil des Werkes darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder einem anderen Verfahren) ohne schriftliche Genehmigung des Verlages reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Printed in Germany
Copyright © 1987 by SYBEX-Verlag GmbH, Düsseldorf

Inhaltsverzeichnis

Einleitung **11**
Über dieses Buch 11
Überblick 12
Vereinbarungen in diesem Buch 14

Teil I
Tutorial 15

1.Kapitel: Einführung in Prolog **17**
Was ist Prolog 17
Zur Entwicklung von Prolog 18
Was an Prolog anders ist 19
Prologs Leistungsfähigkeit 21
Anwendungen in Prolog 24
Expertensysteme 24
Natürliche Abfragesprachen 25
Robotik 26
Spiele und Simulationen 26
Compiler oder Interpreter 27
Eigenschaften von Turbo Prolog 28
Einschränkungen von Turbo Prolog 28

2. Kapitel: Arbeiten mit Turbo Prolog **31**
Starten von Turbo Prolog 31
Die vier Fenster 32
Die sieben Auswahlmöglichkeiten 33
Erstellen eines Beispielprogramms 34
Editieren eines Programms 36
Cursorsteuerung 38
Text einfügen 39
Text löschen 39
Blockoperationen 40
Suchen und Ersetzen 42
Verschiedene Funktionen 42
Hilfsinformationen 43
Verlassen des Editors 43
Dateioperationen: Speichern und Laden 43
Compilieren und Ausführen eines Programms 47

Verändern der Fensterparameter 48
Ändern des voreingestellten Plattenverzeichnisses 49
Ändern des Hintergrundes 50
Wenn Schwierigkeiten auftreten 50

3. Kapitel: Fakten, Objekte und Prädikate 51
Spezifikation von Fakten 51
Objekte in Turbo Prolog 53
Objekte in der Theorie der wissensbasierten Systeme 55
Objekte in prozeduralen Sprachen 56
Das Turbo Prolog-Programm 57
Ein einfaches Turbo Prolog-Programm 60
Übungen 62

4. Kapitel: Variablen in Prolog 65
Die Prolog-Variable 65
Die Verwendung von Variablen 66
Gebundene und freie Variablen 67
Anonyme Variablen 68
Zusammengesetzte Ziele 69
Backtracking 71
Vereinbarungen von Variablen 72
Abschließende Bemerkungen 72
Übungen 73

5. Kapitel: Verwendung von Regeln 75
Regeln 75
Einsatz von Regeln bei Problemlösungen 76
Verwendung von Variablen in Regeln 77
Programmausführung in Prolog 81
Verwendung eines Ablaufprotokolls 82
Unifikation 82
Steuerung des Programmablaufs 84
Standard-Prädikate 85
Das Prädikat not 86
Übungen 87

6. Kapitel: Einfache Ein- Ausgaben 89
Ein Dialogbeispiel 89
Prädikate für Ausgabeoperationen 91
Das Prädikat write 91
Steuerung 92
Ausgabe auf den Drucker 93
Das Prädikat writef 94
Prädikate für Eingabeoperationen 95
Das Prädikat readln 96
Das Prädikat readchar 97

Das Prädikat readint 98
Das Prädikat readreal 98
Die Prädikate inkey und keypressed 99
Verwendung von Fenstern 99
Übungen 102

7. Kapitel: Programmablauf-Steuerung: Durch Scheitern zum Ziel 103
Verwendung des Prädikats fail 104
Ausschluß von Objekten 107
Übungen 108

8. Kapitel: Programmablauf-Steuerung 109

Das Konzept der Rekursion 109
Ein einfaches Beispiel 109
Das Prädikat repeat 110
Eine Anmeldeprozedur 110
Verwendung des Prädikates repeat 112
Grundregeln der Rekursion 114
Zurücksetzen der Rekursionsebenen 116
Übungen 117

9. Kapitel: Programmablauf-Steuerung: Der Schnitt 119
Der Schnitt 119
Beispiel für einen Schnitt 119
Einsatzschwerpunkte für den Schnitt 121
Ein weiteres Beispiel 123
Gemeinsame Verwendung der Prädikate cut und repeat 127
Typen von Schnitten 128
Determinismus 129
Übung 130

10. Kapitel: Arithmetische Operationen 131
Ein einfaches Beispiel 131
Das Gleichheitszeichen 132
Vergleichsoperatoren 134
Operatoren und ihre Stellung 136
Arithmetische Standard-Operatoren 137
Eingebaute mathematische Funktionen 137
Testen von Variablenwerten 140
Verwendung von arithmetischen Prädikaten 140
Zähler 141
Besonderheiten 142
Typumwandlungen 143
Erzeugen von Zufallszahlen 143
Übungen 144

11. Kapitel: Verwendung von zusammengesetzten Objekten **145**
Zusammengesetzte Objekte 145
Ein einfaches Beispiel 145
Gebrauch von zusammengesetzten Objekten 147
Anzahl der Regeln in einem Programm einschränken 148
Verbinden von Informationen zu einem Objekt 148
Unterscheiden zwischen mehreren Objekten 149
Deklaration von zusammengesetzten Objektarten 149
Eingabe von Daten in ein zusammengesetztes Objekt 150
Übungen 151

12. Kapitel: Verwendung von dynamischen Datenbanken **153**
Das medizinische Diagnose-Programm 153
Zwei Typen von Datenbanken 154
Die statische Datenbank 154
Die dynamische Datenbank 156
Typzuweisung für Prädikate dynamischer Datenbanken 157
Verwendung der dynamischen Datenbank 161
Löschen von Fakten 162
Speichern der dynamischen Datenbank 163
Erstellen von Menüstrukturen 164
Übungen 166

13. Kapitel: Arbeiten mit Listen **167**
Listen 167
Deklaration von Listen 168
Unifikation bei Listen 168
Listen von Listen 171
Verwendung von zusammengesetzten Objekten 171
Programmieren mit Listen 171
Listenoperationen 173
Schreiben von Listen 173
Verketten von Listen 173
Umkehren einer Liste 174
Bestimmung eines letzten Elementes einer Liste 174
Bestimmung des n-ten Elements einer Liste 175
Verschiedene Domain-Typen in einer Liste 175
Übungen 176

14. Kapitel: Zeichenkettenoperationen **177**
Zeichenketten in Turbo Prolog 177
Zeichenkettenoperationen 178
Verketten 178
Übungen 184

15. Kapitel: Fenster, Grafik und Sound **185**
Erstellen von Fenstern 185

Anwendung der Fenster-Technik 189
Bildschirmbezogene Ein- und Ausgabe 190
Ein- und Ausgabe von Zeichen 190
Ein- und Ausgabe von Feldern 191
Ein- und Ausgabe von Fenstern 192
Verwendung des Editors im Programm 192
Grafik und Sound 193
Sound 198
Übungen 200

16. Kapitel: Dateiverarbeitung mit Turbo Prolog 201
Einführende Bemerkungen über Dateien 201
Umlenken von Ein- und Ausgabe 202
Öffnen und Schließen von Dateien 203
Prädikate zum Öffnen von Dateien 206
Entleeren der Puffer 207
Direkter Zugriff auf Dateien 207
Verwendung von Hilfsdateien 209
Dateienoperationen in DOS 210
Übungen 213

17. Kapitel: Programmentwicklung in Turbo Prolog 215
Programmierstil und Programmentwurf 215
Programmiertricks und -techniken 216
Verwendung von internen Zielen 218
Compilierung für externen Gebrauch 219
Programmablaufverfolgung 220
Test auf zusammengesetzte Fließmuster 222
Überlaufkontrolle 222
Warnungen und Unterbrechungen 222
Determinismus 223
Erstellen einer Prolog-Bibliothek 225
Modulares Programmieren 225
Anwendung von modularer Programmierung 226
Verwendung von globalen Prädikaten und Domains 227
Aufbau eines Turbo-Prolog-Programms 227
Übungen 228

Teil II
Anwendungen 229

18. Kapitel: Einführung in Expertensysteme 231
Das Expertensystem 231
Aufgaben des Expertensystems 231
Der Wissensverarbeitungs-Fachmann 233
Wissensdarstellung 233
Typen von Expertensystemen 234

Rahmenbasierende Systeme 238
Übungen 239

19. Kapitel: Aufbau eines Expertensystems 241
Festlegen des Fachgebiets und Formulierung der Zielaussage 241
Beschaffung der Fakten 242
Tabellieren der Fakten 245
Bündeln 248
Schreiben des Programms 248
Programmtest 259
Übungen 259

20. Kapitel: Abenteuerspiele und natürliche Abfragesprachen 261
Entwurf eines Geheimnisses 262
Erster Spielentwurf 263
Prolog und natürliche Sprachverarbeitung 265
Kreativität mit Turbo Prolog 268
Ein Beispiel 268
Übungen 273

Anhang A: Installierung von Turbo Prolog 275
Was Sie brauchen 275
Anfertigen von Kopien 277
Drucken der Datei README 278
Erstellen einer Arbeitskopie 279
Anhang B: Verzeichnis der Standard-Prädikate 283
Anhang C: Reservierte Wörter 293
Anhang D: Glossar 295
Anhang E: Literaturverzeichnis 305
Anhang F: Ausgewählte Lösungen 307
Anhang G: Turbo Prolog und andere Prolog-Versionen 319

Einleitung

Bis 1986 war Prolog in erster Linie ein wissenschaftliches Hilfsmittel für Forscher, die sich mit künstlicher Intelligenz beschäftigten. Bücher, Zeitungsartikel und Software über Prolog wurden hauptsächlich von technisch-orientierten Programmierern und professionellen Wissensverarbeitungs-Fachleuten gelesen und angewendet. Die Kosten für die Entwicklung und Verwendung von Produkten künstlicher Intelligenz waren für die meisten Firmen und Entwickler mit einem kleinerem Budget untragbar.

Plötzlich änderte sich das alles. Die Anwender von Personal-Computern zeigten mehr und mehr Interesse an Expertensystemen und künstlicher Intelligenz. In den Regalen der Buchläden, die früher vielleicht einen oder zwei wissenschaftliche Texte über Prolog führten, fanden sich nun Dutzende von Prolog-Büchern. In Zeitschriften wurden ausführliche Artikel über Prolog und Expertensysteme abgedruckt, und es erschien sogar ein spezielles Magazin zu diesem Themenbereich auf dem Markt.

Die meisten Leser fühlten sich allerdings in einem Irrgarten von neuen Begriffen und fremdartigen Programmierungskonzepten allein gelassen. Es gab nur wenige Standards, und die meisten Wissensverarbeitungs-Fachleute veröffentlichten ihre persönlichen Informationen und Programme nur zögernd.

Im Mai 1986 kam der Durchbruch. In diesem Monat erschien von der Firma Borland International ein neuer Prolog-Compiler auf dem Markt, dessen Preis nur rund ein Zehntel von dem seiner Hauptkonkurrenten betrug. Zudem war er einfacher zu handhaben, außerdem schneller als alle anderen und verfügte über viele eingebaute Funktionen, die bei anderen Prolog-Versionen noch extra bezahlt werden mußten. Jetzt konnte praktisch jeder ein Fachmann auf dem Gebiet der Wissensverarbeitung werden.

Über dieses Buch

Dieses Buch wendet sich an diese neuen Anwender von Turbo Prolog. Der Text ist mit vielen Beispielen und Übungen leicht verständlich aufbereitet. Sie

sollten sich die Zeit nehmen, beim Lesen alles mit Ihrem eigenen Turbo Prolog-Programm nachzuvollziehen. Dann werden Sie, nachdem Sie das Buch durchgelesen haben, in der Lage sein, diagnostische Expertensysteme zu schreiben und umfassende Abenteuerspiele zu entwerfen, die Befehle in einer natürlichen Sprache entgegennehmen können.

Ein Wort zur Vorsicht: Prolog ist von Natur aus eine nicht-prozedurale Sprache; d. h. die Ausführung folgt einer Steuerung, die durch die Daten einer Datenbank bestimmt wird, und nicht einer Prozedur, die vom Programmierer definiert wird. Prolog zu erlernen, ist ebenfalls ein nicht-linearer Vorgang. Sie werden beispielsweise länger brauchen, um den Umgang mit Schnitten und Rekursionen (Prologs Ablaufsteuerung) zu erlernen als die Handhabung von Ein- und Ausgabeoperationen zu beherrschen. Auch wenn Sie eins der Themen noch nicht vollkommen im Griff haben, können Sie mit den weiteren Kapiteln und Übungen fortfahren. Später können Sie dann, ebenso wie es ein Prolog-Programm macht, zu den Kapiteln zurückkehren, die Ihnen anfangs Schwierigkeiten bereiteten.

Turbo Prolog wird Ihnen auf jeden Fall mehr Spaß machen als jede andere Computersprache, mit der Sie es bisher zu tun hatten. Mit Ausnahme einiger Besonderheiten (wie die Schnitte) ähnelt die Sprache bemerkenswerterweise dem menschlichen Denkprozeß. Sie ist natürlich und leicht anzuwenden, und umfangreiche Programme können sehr schnell entwickelt werden.

Überblick

Das vorliegende Buch besteht aus zwei Teilen mit insgesamt 20 Kapiteln. Der erste Teil ist ein Lern- und Übungsteil für Prolog, und der zweite beinhaltet die Anwendungen.

Teil I enthält einen intensiven Lernkurs für die Sprache, der zahlreiche Beispiele und Übungen am Ende jedes Kapitels einschließt (ab Kapitel 2). Die 17 Kapitel dieses ersten Teils führen Sie durch das Turbo Prolog-Programm. Mit dem Begriff "Turbo Prolog -spezifisch" in der folgenden Beschreibung ist jeweils gemeint, daß das entsprechende Kapitel Gesichtspunkte über Turbo Prolog anschneidet, die sich von anderen Prolog-Versionen wesentlich unterscheiden.

Kapitel 1 stellt eine Einführung in Turbo Prolog dar. Es beschreibt, worin sich Prolog von anderen Sprachen unterscheidet. Kapitel 2 gibt einen schnellen Überblick über das Erstellen, Compilieren und Ausführen eines Turbo Prolog-Programms. Die Kapitel 3, 4 und 5 enthalten die Grundlagen zu Prolog.

Dabei stellt das Kapitel 3 Fakten und Ausdrücke vor. Die Erörterung von Argumenttypen in diesem Kapitel ist Turbo Prolog-spezifisch. Kapitel 4 beschreibt die Verwendung von Variablen, und Kapitel 5 führt in den Gebrauch von Regeln ein. Kapitel 6 behandelt Ein- und Ausgaben und ist ebenfalls Turbo Prolog-spezifisch.

Die Kapitel 7, 8 und 9 zeigen die Techniken der Ablaufsteuerung in Prolog. Kapitel 7 erläutert das Prädikat *fail*, Kapitel 8 führt die Rekursion ein, und Kapitel 9 behandelt den Schnitt (/). Kapitel 10, das Turbo Prolog-spezifisch ist, beschreibt arithmetische Operationen.

In Kapitel 11 werden zusammengesetzte Objekte beschrieben, die zu den komplexen Strukturen gezählt werden. Kapitel 12 behandelt dynamische Datenbanken. Kapitel 13 erläutert die Listen-Struktur, und Kapitel 14 stellt die Prädikate für Zeichenkettenoperationen vor, die wieder Turbo Prolog-spezifisch.

Die Kapitel 15 und 16 untersuchen fortgeschrittenere Themen wie Fenster, Grafik, Sound und Dateiverarbeitung. Alle diese Punkte sind für Turbo Prolog spezifisch und haben wenig Gemeinsamkeiten mit anderen Sprachen. Kapitel 15 behandelt sowohl Fenster, Grafik und Sound als auch einige spezielle Ausgabe-Prädikate. Kapitel 16 gibt einen Überblick über Dateioperationen in Turbo Prolog.

Kapitel 17 vermittelt eine Übersicht über Programmiertechniken und Ablaufmechanismen beim Gebrauch von Turbo Prolog. Es enthält außerdem Programmiertips und erörtert Compiler-Übersetzungsanweisungen, Ablaufverfolgung, Fehlersuche und modulares Programmieren.

Teil II dieses Buches ist ein Anwendungsleitfaden. Kapitel 18 gibt eine Einführung in Expertensysteme. Kapitel 19 zeigt einen vollständigen Entwurf eines diagnostischen Expertensystems zur Diagnose von Kinderkrankheiten – ein System, das als fortlaufendes Beispiel in Teil I entwickelt wurde. Kapitel 20 behandelt Spiele und natürlich-sprachliche Schnittstellen. Ebenso wie Teil I beinhaltet auch Teil II Übungen am Ende jedes Kapitels.

Das Buch umfaßt ferner sieben Anhänge. Anhang A erläutert, wie Sie Sicherungskopien von Ihren Programmdisketten machen und Turbo Prolog auf verschiedenen Computersystemen installieren können. Anhang B faßt die (eingebauten) Standard-Prädikate zusammen; das schließt auch die Elemente ein, die in anderen Sprachen als Befehle und Funktionen bezeichnet würden.

Anhang C gibt eine Aufstellung der reservierten Wörter, die bei Turbo Prolog nicht als Namen verwendet werden dürfen. Anhang D ist ein Glossar. Anhang

E stellt Bücher, Zeitschriften und andere Quellen zur weiteren Vertiefung vor. Anhang F gibt Lösungen zu ausgewählten Übungen an. Anhang G vergleicht Turbo Prolog mit anderen Prolog-Versionen.

Vereinbarungen in diesem Buch

Da es bis jetzt noch wenige Normen im Bereich von Prolog und künstlicher Intelligenz gibt, ist es schwierig, einheitliche Bezeichnungen in Büchern und Artikeln zu diesen Themen zu finden. Begriffe variieren in ihrer Bedeutung, und es gibt leider keinen allgemein akzeptierten Prolog-Standard. Im allgemeinen wird ein Turbo Prolog-Programm nicht ohne größere Änderungen unter anderen Prolog-Versionen laufen. Wir haben uns aber bemüht, in diesem Buch so weit wie möglich die Definitionen und die Terminologie des Turbo Prolog-Handbuchs zu übernehmen und dabei größere Abweichungen von der in der künstlichen Intelligenz üblichen Terminologie kenntlich zu machen.

Turbo Prolog ist nicht das gleiche wie Prolog. Es ist sowohl eine Untermenge als auch eine Obermenge des Prolog-Standards, wie er ursprünglich in der Veröffentlichung von Clocksin und Mellish (C&M, siehe Anhang E) vorgeschlagen wurde. Wir verwenden daher im vorliegenden Buch den Begriff *Prolog*, wenn wir uns auf diesen C&M-Standard beziehen, und *Turbo Prolog*, um spezielle Eigenschaften von Turbo Prolog herauszustellen.

TEIL 1

Tutorial

Kapitel 1

Einführung in Prolog

Für viele Computer-Freunde ist Prolog die geheimnisvollste der Programmiersprachen. Anders als fast alle anderen Programmiersprachen unterstützt Prolog formale symbolische Schlußfolgerungen, die den Computer deshalb "intelligent" erscheinen lassen, da er quasi die menschliche Sprache versteht und medizinische Erkrankungen diagnostizieren oder, Routineaufgaben ohne eine menschliche Vorgabe ausführen kann.

Aber genau wie bei den anderen Sprachen ist auch ein Computer, der mit Prolog arbeitet, nicht aus sich selbst heraus intelligent. Das im Computer gespeicherte Wissen muß ursprünglich von einem Programmierer eingegeben werden. Prolog bietet allerdings einige Vorteile dadurch, daß es bei der Lösung einiger Problemtypen wesentlich effizienter arbeitet als andere Sprachen. Man kann vielleicht sagen, daß ein Computer mit Prolog in dem Sinne "intelligent" ist, daß er diese Probleme in derselben Art und Weise anpackt, wie es ein Mensch tun würde.

Bevor wir nun zu Turbo Prolog kommen, wollen wir noch einen kurzen Überblick über die Sprache und ihre Möglichkeiten geben.

Was ist Prolog?

Der Name Prolog rührt von dem Ausdruck "PROgramming in LOGic" her. Die Sprache ist ursprünglich 1972 von Alain Comerauer und P. Roussel in Frankreich an der Universität von Marseille entwickelt worden.

Prolog ist einmalig in seiner Möglichkeit, Fakten und Folgerungen aus anderen Fakten abzuleiten (formale Schlußfolgerungen zu ziehen). Um Prologs Fähigkeiten bei einer Problemlösung auszuschöpfen, muß der Anwender das Problem in einer logisch zusammenhängenden und formalen Art und Weise beschreiben, indem er Fakten und Kenntnisse über das Problem vorlegt und ein Ziel angibt. Der Computer verwendet diese Informationen, um das anvisierte Ziel zu erreichen, indem er sich selbst dafür eine Prozedur erstellt. In den Kapiteln 3 und 4 werden wir einige spezielle Beispiele untersuchen, die das Prinzip der formalen Schlußfolgerungen in Turbo Prolog verdeutlichen.

Zur Entwicklung von Prolog

In den Siebzigerjahren wurde Prolog in Europa im Zusammenhang mit Anwendungen in künstlicher Intelligenz bekannt. In den USA blieb Prolog jedoch eine verhältnismäßig unbedeutende Computersprache. Dort bevorzugten Forscher die Sprache LISP für Anwendungen in künstlicher Intelligenz. Man hielt LISP für diese Art von Anwendungen für geeigneter, obwohl es schwerer zu lernen und anzuwenden ist als Prolog.

In dieser Anfangszeit waren sowohl Prolog als auch LISP sehr langsam in der Ausführung und verbrauchten große Mengen an Speicherplatz. Außerdem mußten die Benutzer über beträchtliche Programmier-Erfahrungen verfügen, um Prolog- oder LISP-Programme schreiben zu können. Ärzte, Rechtsanwälte, Ingenieure und andere, die am meisten Bedarf für Expertensysteme gehabt hätten, befanden, daß die Komplexität der Sprache die Verwendung von Expertensystemen auf Einsätze in Universitäten, Regierungs- und juristischen Forschungsinstituten beschränkte.

Aber mit der Verbreitung des Mikrocomputers kamen auch bald Prolog-Versionen für diesen Rechner auf den Markt. Diese nahezu gleichbleibenden Ausführungen waren langsam, und, da Prolog ausreichend viel Speicher und einen schnellen Prozessor benötigt, von sehr beschränktem Nutzen. Schließlich wurden Compiler entwickelt, um das Geschwindigkeitsproblem abzuschwächen, aber Prolog blieb immer noch eine Sprache mit beschränkter Einsatzfähigkeit, und nur wenige waren bereit, Geld für die Entwicklung von leistungsstärkeren Prolog-Versionen auszugeben.

Im Jahre 1981 änderte sich bei der ersten internationalen Konferenz für Systeme der fünften Generation in Tokio plötzlich die Situation. Die Japaner hatten große Probleme, zu dem schnell wachsenden Computermarkt in den USA konkurrenzfähig zu bleiben. Sie stellten einen neuen zukunftsweisenden nationalen Technologie-Plan auf und kündigten an, daß sie die gegenwärtige Technologie mit neuer Hard- und Software für die Neunzigerjahre überspringen würden. Die Computersprache für diese neuen Systeme sollte Prolog sein.

Als Folge dieser japanischen Initiative wurde Prolog für Entwickler in aller Welt wieder interessant. Die heutigen Prolog-Versionen verfügen über Eigenschaften und Möglichkeiten, die die der ehemaligen weit übertreffen. Sie sind meist einfacher in der Anwendung, kosten weniger als frühere Versionen und sind schneller.

Prolog ist sowohl auf Großrechnern als auch auf Mikrocomputern oft die beste Sprache, wenn es um Anwendungen geht, an denen formale oder symbolische Schlußfolgerungen beteiligt sind.

Was an Prolog anders ist

Nahezu alle Sprachen, die in den letzten Jahrzehnten für Computer entwickelt worden sind, waren sogenannte prozedurale Sprachen (verfahrensorientierte Sprachen). Beispielsweise gehören FORTRAN, COBOL, C, BASIC, dBASE III und Pascal zu diesen prozeduralen Sprachen, bei deren Verwendung immer zuerst ein Algorithmus festgelegt werden muß, um das Problem zu lösen. Dann wird ein Programm erstellt, das die prozedurale Sprache verwendet, um eine Prozedur zu implementieren. Das Programm kann diese Prozedur beliebig oft mit verschiedenen Eingabedaten ausführen. Es wird bei der Lösung jedes Problems, das die Verwendung dieser Prozedur erfordert, immer schneller und zuverlässiger als ein Mensch sein, aber es bleibt auf die Probleme beschränkt, die diese einzige Prozedur umfaßt.

Wir wollen beispielsweise annehmen, daß eine Kreditbank häufig von Leuten aufgesucht wird, die Geld leihen wollen, um ihr Geschäft zu vergrößern. Die Bank prüft die aktuelle Situation, die bisherige Entwicklung und die Zukunftschancen des jeweiligen Betriebs. Auf der Basis dieser Daten führt der Kreditsachbearbeiter viele Berechnungen durch und entscheidet dann, ob die Bank diesem Betrieb Geld leihen soll.

Der Berechnungsteil dieses Entscheidungsprozesses ist ein objektiver Vorgang, der immer gleich bleibt. Ein Programmierer kann diesen Vorgang für die Bank analysieren und dann ein Programm schreiben, das diesen Berechnungsvorgang durchführt, und den gesamten Ablauf dadurch wesentlich effizienter und zuverlässiger gestalten.

So können komplexe und schwierige Fälle auch in kleineren Außenstellen der Bank abgewickelt werden, in denen es aufgrund des Fehlens entsprechend qualifizierter Mitarbeiter bislang nicht ohne Rücksprache mit einer Hauptgeschäftsstelle möglich war, derartige Fälle selbständig zu bearbeiten.

Prozedurale Sprachen unterscheiden zwischen einem Programm und den Daten, die es verwendet. Der Ablauf und die Kontrollstrukturen, die ein Programm benutzt, werden von einem Programmierer festgelegt. Das Programm verarbeitet die Daten entsprechend der festgelegten Prozedur. Davon unterscheidet sich die Entwicklung eines Prolog-Programms wesentlich.

Es ist sogar besser, daß, wenn man versucht Prolog zu lernen, und man bereits in herkömmlichen Sprachen programmiert hat, man sein ganzes Wissen über Programmierung vergißt. Turbo Prolog wie alle anderen Implementationen von Prolog ist eine Objekt-orientierte Sprache. Es verwendet keine Prozeduren und eigentlich auch kein Programm. (Es ist dennoch üblich, die Syste-

me, die in Prolog geschrieben werden, als Programme zu bezeichnen, und daran werden wir uns auch in diesem Buch halten.)

Prolog verwendet lediglich Daten über Objekte und ihre Beziehungen (Relationen) zueinander. Es handelt sich hier um eine symbolische Verarbeitung. LISP und Prolog sind die bekanntesten Objekt-orientierten Sprachen.

Bei Prolog muß der Anwender ein Ziel, auch *goal* genannt, formulieren (ein Problem oder ein Endergebnis), und der Computer muß sowohl die Prozedur als auch die Lösung erstellen.

Ein Prolog-Programm besteht also aus einer Ansammlung von Daten oder Fakten und Beziehungen zwischen diesen Fakten. Man kann auch sagen, das Programm ist eine Datenbank.

Durch die Eigenschaften dieser Prolog-Datenbank eignet sich die Sprache ideal zur Lösung unstrukturierter Probleme, für die ein Lösungsweg unbekannt ist. Was über das Problem bekannt ist, wird als Daten gespeichert.

Der Anwender definiert ein Ziel oder eine Hypothese, und das Programm versucht mit Hilfe formaler Schlußfolgerungen, diese auf den bekannten Daten basierende Hypothese zu bestätigen oder zu widerlegen.

Da Prolog kein Programm benutzt, benötigt es auch keine Programmierer. Fachleute für Wissensverarbeitung entwerfen und realisieren die Systeme, die Prolog verwenden.

Der Wissensverarbeitungs-Fachmann bekommt auf einem bestimmten Gebiet Informationen von Experten, wobei er von der subjektiven Denkweise des menschlichen Gehirns abstrahiert und ein objektives System formaler Schlußfolgerungen festlegt, das von einer Prolog-Struktur unterstützt werden kann.

Das Bemerkenswerteste an Prolog ist, daß es ermöglicht, mit Hilfe des Computers Dinge zu realisieren, die mit anderen Sprachen nur unvollkommen oder gar nicht bewerkstelligt werden können.

Sie werden jedoch feststellen, daß Prolog uneffizient arbeitet, wenn Sie es zur numerischen oder Zeichenkettenverarbeitung mit einer durch eine bestimmte Aufgabenstellung vorgebenenen prozeduralen Lösung verwenden.

Die Eigenschaften prozeduraler und Objekt-orientierter Sprachen sind in Tabelle 1.1 zusammengefaßt.

Prozedurale Sprachen (BASIC, COBOL, Pascal)	Objekt-orientierte Sprachen (Prolog, LISP)
verwenden vorab definierte Prozeduren, um Probleme zu lösen	verwenden Heuristiken, um Probleme zu lösen
sehr wirksam bei numerischer Verarbeitung	sehr wirksam bei formalen Schlußfolgerungen
Systeme werden von Programmierern erstellt und gepflegt	Systeme werden von Wissensverarbeitungs-Fachleuten (knowledge engineers) entwickelt und gepflegt
verwenden strukturierte Programmierung	interaktive und zyklische Entwicklung

Tabelle 1.1: Objekt-orientierte und prozedurale Sprachen

Prologs Leistungsfähigkeit

Die Leistungsfähigkeit von Prolog beruht auf seiner Fähigkeit, Fakten aus anderen Fakten abzuleiten. Das unterscheidet es wesentlich von einer Sprache, die in erster Linie ein numerischer Prozessor ist.

Ein solcher numerischer Prozessor kann beispielsweise die monatliche Rate für einen Autokredit berechnen. Die Höhe dieser monatlichen Zahlung ist eine neue Größe; sie wurde nicht als Datenwert vom Anwender eingegeben, sondern aufgrund einer zuvor definierten Prozedur ermittelt.

Stellen Sie sich hingegen eine intelligente Rechtschreibhilfe vor, die das Wort erraten kann, das Sie versuchen zu buchstabieren. Der numerische Prozessor würde bei dieser Aufgabe völlig versagen. Wörter haben eine symbolische Bedeutung, die auf dem jeweiligen Zusammenhang beruht; eine intelligente Rechtschreibhilfe muß also auf den Zusammenhang achten und ihre Schlüsse aufgrund der symbolischen Beziehungen zwischen den Wörtern ziehen.

Wenn beispielsweise das Wort *wir* anstelle von *wird* geschrieben wird, kann nur eine Analyse des Textzusammenhangs diesen Schreibfehler aufdecken; die meisten gebräuchlichen Prüfprogramme würden das Wort als richtig zu-

lassen. Mit Prolog wird eine symbolische Verarbeitung ermöglicht. Das Wort *wir* ist ein Pronomen, und wird ist ein Verb.

Ein Rechtschreib-Prüfungssystem in Prolog kann den Fehler daran erkennen, wie das Wort im Satz verwendet wird.

Prozedurale Sprachen verwenden Algorithmen. Algorithmen sind objektive Prozeduren, die in ihrer Folge zu einer Lösung führen. Objekt-orientierte Sprachen wie Prolog verwenden heuristische Verfahren.

Heuristiken sind Faustregeln, die dem Erreichen eines Ziels dienen. Sie garantieren nicht immer eine Lösung und geben auch nicht immer die effizienteste an. Dennoch sind sie nützlich, wenn es keinen Algorithmus gibt.

Bei einem Schachzug beispielsweise werden Sie auch nicht alle möglichen Alternativen durchdenken oder eine festgelegte Prozedur verwenden, um den nächsten Zug zu bestimmen. Sie werden nach gewissen Regeln vorgehen, die auf Ihrem Erfahrungs- und Wissensschatz mit diesem Spiel beruhen.

Ihre Entscheidung mag die richtige sein, aber es kann vielleicht noch eine bessere Möglichkeit geben. Sie verwenden also eine heuristische Methode bei der Wahl ihres Zugs.

In der menschlichen Denkweise wird ein zu lösendes Problem in kleinere Einheiten aufgeteilt, und die Lösungen zu diesen Detailproblemen bilden eine Reihe von Zwischenergebnissen. Jedes Zwischenergebnis ist ein Schritt auf das Endziel hin.

Angenommen, ein Tonband auf Ihrer Stereoanlage ist beim Abspielen nur auf einem Lautsprecher zu hören. Um dieses Problem zu lösen, könnten Sie zuerst das Stereogerät vom Tonbandgerät auf den UKW-Eingang umschalten. Wenn nun der Stereoklang über beide Kanäle kommt, wissen Sie, daß der Band-Eingang nicht ordnungsgemäß funktioniert.

Als nächstes könnten Sie es mit einem anderen Band versuchen, um herauszufinden, ob das zuerst verwendete Band nicht in Ordnung ist. Schließlich könnten Sie den Tonkopf reinigen.

Sie führen also eine Reihe von Schritten aus, um die Ursache des Problems einzukreisen. Jeder Schritt ist eine Heuristik (Abb. 1.1).

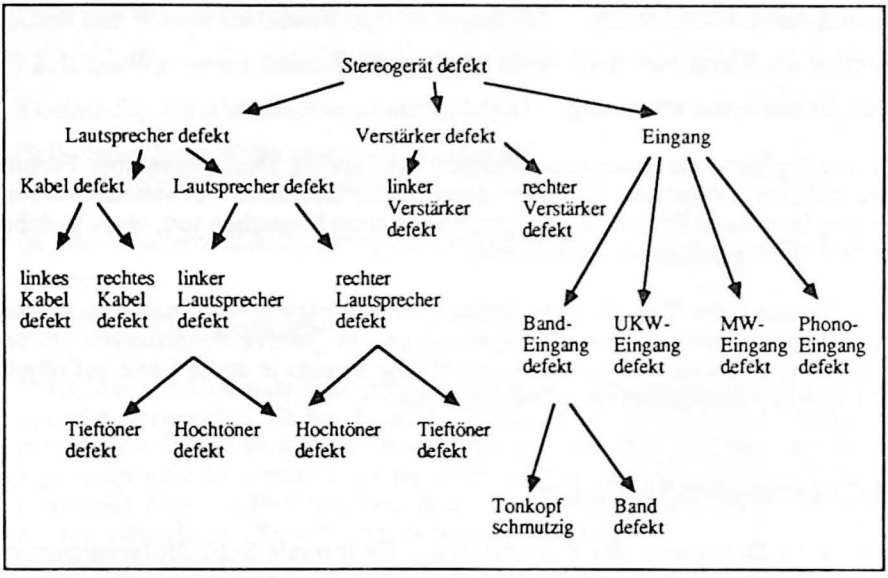


Abb. 1.1: Einkreisen eines Problems

In gleicher Weise erstellen Sie Prolog-Systeme, indem Sie Heuristiken anstelle von Algorithmen verwenden. Der Problemraum besteht aus Knoten und Verbindungen. Jeder Knoten stellt ein Zwischenziel oder einen Schritt auf dem Weg zum Endziel dar.

Die Verbindungen sind die Operationen, mit denen Sie sich zwischen den Knoten bewegen. In einem prozeduralen Programm müssen Sie einen speziellen Weg durch diesen Problemraum festlegen.

Bei Prolog brauchen Sie nur den Problemraum zu definieren. Dabei stellen Sie jedoch bestimmte Heuristiken über das tatsächliche Problem auf, das er darstellt. Das sich ergebende Prolog-System sollte einen Dialog mit dem Anwender führen und in derselben Art und Weise Schlüsse ziehen, wie es ein Mensch bei der Lösung eines Problems auch tun würde. Ein Dialog zwischen einem Anwender und Prolog könnte beispielsweise folgendermaßen aussehen:

Kommt der Klang beim Band durch beide Lautsprecher? – *Nein.*

Kommt der Klang bei UKW durch beide Lautsprecher? – *Ja.*

Versuchen Sie es mit einem anderen Band.

Kommt der Klang nun durch beide Lautsprecher? – *Nein.*

Reinigen Sie den Tonkopf.

Kommt der Klang nun durch beide Lautsprecher? – *Ja.*

Der Tonkopf war schmutzig.

Prolog-Systeme zu entwerfen, erfordert eine andere Denkweise über Probleme und ihre Lösungen, als wenn prozedurale Sprachen verwendet werden. Prolog bearbeitet Probleme mehr so, wie es auch Menschen tun, obwohl dabei nicht alle Möglichkeiten offenstehen.

Das menschliche Gehirn kann formale Schlußfolgerungen, numerische Berechnungen, Analogschlüsse, Eingebung und subjektive Folgerungen bei der Lösung von Problemen einsetzen. Bei Prolog ist man in erster Linie auf objektive formale Schlußfolgerungen beschränkt.

Anwendungen in Prolog

Prolog ist für nahezu alle Anwendungen, die formale Schlußfolgerungen erfordern, geeignet. Das schließt Anwendungen in Expertensystemen, natürlichen Abfragesprachen, Robotik, Spielen und Simulationen ein.

Expertensysteme

Expertensysteme sind Programme, die unter Verwendung eines Schlußfolgerungs-Mechanismus, der sogenannten Inferenzmaschine, das Verhalten eines menschlichen Experten bei der Lösung eines Problems auf einem Spezialgebiet simulieren können.

Expertensysteme können beraten, diagnostizieren, analysieren und kategorisieren, indem sie eine zuvor definierte Wissensbasis benutzen. Diese Wissensbasis ist eine Ansammlung von Regeln und Fakten, die meistens unter Prolog erstellt wird.

Bei der Verwendung eines Expertensystems legt der Anwender zunächst ein Ziel fest. Das System versucht dann, dieses Ziel durch Stellen von Fragen und unter Verwendung seines intern gespeicherten Wissens zu erreichen.

In einigen Fällen kann das System auch Multiple-Choice-Fragen stellen. Stellen Sie sich beispielsweise ein Expertensystem für die Reparatur eines Geschirrspülers vor:

Läuft das Wasser im Geschirrspüler richtig ab?	–	*Nein.*
Läuft das Wasser zur falschen Zeit ab?	–	*Ja.*
Können Sie den Abflußschlauch untersuchen?	–	*Ja.*
Befindet sich der Abflußschlauch in Höhe der oberen Kante des Geschirrspülers?	–	*Ja.*
Ist eine Wasserrücklaufsperre in der Abflußleitung?	–	*Nein.*

Natürliche Abfragesprachen

Jeder, der mit Computern zu tun hat, weiß, daß diese einen anderen Typ von Sprache verwenden als der Mensch. Warum können wir dem Computer eigentlich die Probleme nicht in unserer Sprache mitteilen und ihn diese in seine eigene Sprache umwandeln lassen, wenn er doch so intelligent ist? Wenn der Computer dann ein Problem bearbeitet hat, warum sollte er die Antwort nicht in eine menschliche Sprachform zurückverwandeln?

Bei Prolog hat man begonnen, den Computer genau dorthin zu führen. Informationen über eine menschliche Sprache werden mit Hilfe von Prolog in Regeln und Fakten ausgedrückt. Der Computer kann dann einen Dialog mit dem Anwender beginnen (oft auch Beratung genannt), wobei er Fragen über das Problem stellt und die Antworten des Anwenders auswertet.

Oft sind solche *natürlich-sprachlichen Prozessoren* Teil von Expertensystemen, die es auch einem technisch weniger versierten Anwender ermöglichen, Probleme zu beschreiben und sie zu lösen. Betrachten wir das Beispiel eines Expertensystems mit einer Schnittstelle für eine natürliche Abfragesprache, das Hilfe bei der Reparatur eines Geschirrspülers leisten soll. Der Dialog mit dem Computer könnte folgendermaßen lauten:

Welches Problem gibt es mit dem Geschirrspüler?
Das Wasser läuft ab, bevor der Waschvorgang abgeschlossen ist.

Können Sie den Abflußschlauch untersuchen?
Ja.

Befindet sich der Abflußschlauch in Höhe der oberen Kante des Geschirrspülers?

Fast ganz oben

Ist eine Wasserrücklaufsperre in der Abflußleitung?
Ich weiß nicht, was das ist.

Beachten Sie den Unterschied zwischen diesem und dem vorigen Beispiel. Die Antworten des Anwenders sind nicht auf Ja und Nein beschränkt. Dieser Dialog ist natürlicher und *menschlicher.*

Obwohl natürlich-sprachliche Prozessoren für die heutigen Mikrocomputer noch zu viel Speicher und Verarbeitungsaufwand erfordern, ist das Konzept mit schnelleren Prozessoren durchaus realisierbar, wenn ausreichend Speicherplatz zur Verfügung steht.

Es ist möglich, daß Entwickler, sobald schnellere Prozessoren und noch mehr Speicherplatz verfügbar sein werden, das Wissen, das bei Lösungen von Problemen eines bestimmten Typs gebraucht wird, auf eine kleine Kunststoffdiskette, CD-ROM genannt, aufbringen. Dieses CD-ROM wird dann vermutlich für jedermann erschwinglich sein.

Robotik

Robotik ist ein Zweig der künstlichen Intelligenz, der sich damit befaßt, Computern das Erfassen und Handhaben von Objekten in ihrer Umgebung zu ermöglichen. Die Robotik-Forschung beschäftigt sich in erster Linie mit der Untersuchung und Entwicklung von Sensor-Systemen, Manipulatoren, Steuerungen und heuristischen Verfahren zur Lösung von Objekt- und Raum-orientierten Umgebungsproblemen.

Prolog vereinfacht die Entwicklung von Robotik-Steuerprogrammen, die Eingabedaten von Sensoren entgegennehmen, um Manipulatoren zu steuern. Ein Roboter kann dann Entscheidungen aufgrund formaler Schlußfolgerungen ziehen, ähnlich wie es ein Mensch tun würde.

Spiele und Simulationen

Prolog ist besonders gut für Spiele und Simulationen geeignet, bei denen formale Schlußfolgerungen eine Rolle spielen. Die meisten Spiele und Simulationen verwenden einen Satz logischer Regeln, die das Spiel oder die Aktion steuern, und dies paßt genau zur Prolog-Programmierung. Mit Prolog ist es möglich, verschiedene heuristische Verfahren einer bestimmten Steuerungstaktik versuchsweise gegenüberzustellen.

Viele klassische Spiele, wie z. B. ELIZA, Türme von Hanoi, das "Achtdamenproblem beim Schach" und einige Abenteuerspiele sind schon in Prolog realisiert worden.

Compiler oder Interpreter

Computer-Programme werden als Quellcode in einer bestimmten Sprache geschrieben. Ein *Compiler* oder *Interpreter* wird zur Umwandlung des Programm-Quelltextes in eine für den Computer verständliche Form gebraucht. Jede Sprache hat für sie spezifische Compiler oder Interpreter.

Wenn die Sprachumwandlung mit Hilfe eines Interpreters erfolgen soll, so muß dieser gleichzeitig mit dem Quellcode im Computer-Speicher vorhanden sein. Jedesmal, wenn eine Programmzeile ausgeführt wird, liest sie der Interpreter, übersetzt sie und führt sie dann aus. Deshalb arbeiten Interpreter natürlich ziemlich langsam.

Ein anderer Nachteil ist, daß sie wertvollen Speicherplatz belegen, da sie zusammen mit dem Quellcode im Speicher stehen müssen. Ein dritter Nachteil ist, daß der Anwender eine Kopie des Interpreters benötigt, um das Programm laufenzulassen.

Interpreter bieten jedoch auch einen Vorteil. Der Interpreter steht während der Ausführung jederzeit zur Verfügung, und Interpreter-Anweisungen können aus dem Programm heraus angesprochen werden; Programme können unterbrochen, untersucht, verändert und erneut ausgeführt werden. Es können sogar Daten bei einer Programmunterbrechung geändert werden, oder die Programmausführung kann an der Stelle wieder aufgenommen werden, an der sie unterbrochen wurde.

Im Gegensatz dazu wandelt ein Compiler den Quellcode in ein Programm um, das selbständig laufen kann. Danach wird der Compiler nicht länger benötigt, bis das Programm verändert wird; der Anwender arbeitet nur noch mit dem compilierten Programm. Dieses ist in der Ausführung wesentlich schneller, und Speicherplatz wird nur für das Programm und für Daten gebraucht.

Turbo Prolog ist ein echter Compiler, der den Quellcode in eine ausführbare Form compiliert. Sie können Turbo Prolog zum Compilieren von Programmen verwenden, die dann ohne Prolog laufen, so daß Sie Ihre Programme verkaufen könnten, ohne Tantiemen an die Firma Borland International zahlen zu müssen. Sie verfügen außerdem über die Ausführungsgeschwindigkeit eines echten Compilers.

Eigenschaften von Turbo Prolog

Turbo Prolog ist ein Prolog-Compiler, der für den IBM PC und kompatible Computer von der Firma Borland International entwickelt und auf den Markt gebracht wurde. Es enthält die meisten der Merkmale des Prologs von Clocksin und Mellish, das als das Standard-Prolog gilt. Aus der Sicht des Anwenders verhält es sich wie ein Compiler. Die compilierten Programme sind in der Ausführung schneller als jedes andere Prolog-Programm, das für den IBM PC und kompatible Computer auf dem Markt ist.

Turbo Prolog bietet die folgenden Möglichkeiten:

Sie können eigenständige Programme erzeugen, die auch auf Computern ohne Prolog laufen. Diese Programme können an Anwender verkauft oder weitergegeben werden, ohne daß irgendwelche Tantiemen an Borland International gezahlt werden müßten.

- Ein vollständiger Vorrat an Standard-Prädikaten für Funktionen wie Zeichenkettenverarbeitung, Direktzugriff auf Dateien, Cursorsteuerung, Grafik, Fenstertechnik und Sound ist verfügbar.

- Eine praktische Schnittstelle zu anderen Sprachen steht zur Verfügung, die die Einbindung von Programmteilen in einer prozeduralen Sprache gestattet.

- Variablen werden deklariert, um eine bessere Kontrolle bei der Programmentwicklung zu gewährleisten.

- Es wird sowohl Festkomma- als auch Fließkomma-Arithmetik unterstützt.

- Ein eingebauter Editor erleichtert die Programmentwicklung, Compilierung und Fehlersuche wesentlich.

Einschränkungen von Turbo Prolog

In gewisser Weise spiegelt Turbo Prolog die Strukturierung der prozeduralen Sprachen wider. Die Erfordernis der Variablendeklaration und die Einteilung des Programms in Abschnitte (siehe Kapitel 3) bedeutet eine gewisse Einschränkung für die symbolische Verarbeitung von Turbo Prolog, wie es in den herkömmlichen Prolog-Versionen nicht der Fall ist.

Die aktuelle Version von Turbo Prolog unterstützt keinen virtuellen Speicher. Sonst wäre die Programmgröße nur noch durch die Plattenkapazität beschränkt. So ist die Größe eines mit Turbo Prolog entwickelten Systems jedoch durch die Menge des verfügbaren Speicherplatzes beschränkt.

Sie können mit Direktzugriffsdateien arbeiten, um diese Einschränkung bis zu einem gewissen Grad zu umgehen, aber dabei bleibt die Speicherplatzkontrolle immer in der Verantwortung des Anwenders, im Gegensatz zu Prolog-Versionen mit virtueller Speicherverwaltung wie etwa Arity Prolog, das den Speicher automatisch verwaltet.

Turbo Prolog ist wie alle Prolog-Versionen unbrauchbar für numerische Berechnungen. Sie werden sehen, daß Sie im Vergleich zu den prozeduralen Sprachen nur auf wenig elegante Art und Weise prozedurale Programmstücke einschieben können.

Kapitel 2

Arbeiten mit Turbo Prolog

Bevor wir die Sprache Prolog näher untersuchen, wollen wir zunächst ein einfaches Programm mit Turbo Prolog erstellen, verändern und ausführen. Dieses Kapitel gibt eine Einführung in die grundlegenden Ablaufmechanismen bei der Verwendung von Turbo Prolog und zeigt den Weg von der Programmerstellung über das Editieren bis zum Ausführungsprozeß.

Es macht nichts, wenn Sie danach noch nicht vollständig verstanden haben, wie Prolog arbeitet. Das Kapitel will Ihnen lediglich die Anwendung der wichtigsten Turbo Prolog-Befehle zeigen, die für das Erstellen eines ersten Programms ausreichen.

Starten von Turbo Prolog

Um Turbo Prolog von der Festplatte aus zu starten, müssen Sie das Unterverzeichnis von Prolog aufrufen und dann

```
C>PROLOG <Return>
```

eingeben. Wenn Sie ohne Festplatte arbeiten, legen Sie eine Arbeitsdiskette in das Laufwerk A und tippen das gleiche ein. (Informationen über die Installation finden Sie in Anhang A.)

Turbo Prolog wird geladen, und auf dem Bildschirm erscheint eine Copyright-Meldung und die Aufforderung, die Leertaste zu drücken (Press the SPACE bar). Das Drücken der Leertaste beendet die Startprozedur.

Nun werden auf dem Bildschirm vier Fenster angezeigt, und am oberen Bild-

schirmrand erscheint ein Menübalken mit sieben Auswahlmöglichkeiten (Abb. 2.1). Diese Anzeige wollen wir nun genauer betrachten.

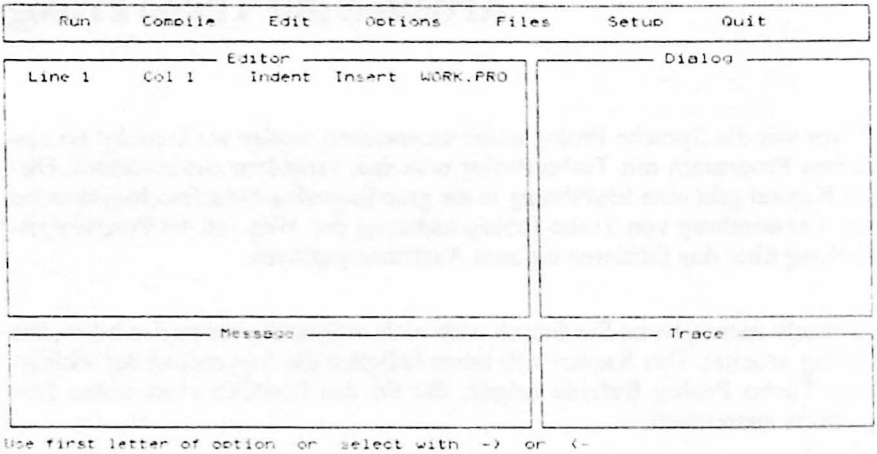


Abb. 2.1: Starten des Prolog-Programms

Die vier Fenster

Die vier Fenster tragen die Bezeichnungen: Editor, Dialog, Message und Trace. Das Editor-Fenster dient zum Erstellen und Editieren von Programmen. Während der Ausführung eines Prolog-Programms werden die Ergebnisse im Dialog-Fenster ausgegeben. Ebenso muß die Eingabe von Daten während der Programmausführung über das Dialog-Fenster erfolgen. Diese Fenster stellt also den Kommunikationsbereich mit dem gerade ablaufenden Prolog-Programm dar. Das Message-Fenster für die Prolog-Systemmeldungen informiert Sie über die Programmaktivitäten sowie über Fehlerzustände, und das Trace-Fenster hilft Ihnen während der Testphase des Programms, Fehler zu finden und zu beseitigen.

Alle vier Fenster können beliebig auf dem Bildschirm verschoben werden. Außerdem können sie verkleinert, vergrößert oder überlagert werden. Sie können die Fenster an Ihre speziellen Bedürfnisse anpassen und dann diese neue Zusammenstellung für eine spätere Verwendung abspeichern. Doch darauf kommen wir noch einmal zurück.

Die sieben Auswahlmöglichkeiten

Am oberen Bildschirmrand werden in einem Menübalken sieben Auswahlpunkte präsentiert: Run, Compile, Edit, Options, Files, Setup, Quit. Dabei ist die Auswahl Run voreingestellt, was uns allerdings augenblicklich noch nicht interessiert, da kein Programm geladen ist.

Um einen Punkt anzuwählen, der dann in einem hellen Kästchen erscheint, können Sie entweder mit Hilfe der Pfeiltasten zu dem betreffenden Wort gehen und dann <Return> drücken oder den ersten Buchstaben der jeweiligen Auswahlmöglichkeit eintippen. Um beispielsweise Prolog zu verlassen, können Sie das helle Kästchen auf das Wort Quit bringen und <Return> drücken oder den Buchstaben Q eintippen. Einige der Auswahlpunkte verfügen noch über zusätzliche Menüs. Wenn Sie zum Beispiel den Buchstaben F drücken (zur Auswahl der Option Files), wird ein weiteres Menü erscheinen (Abb. 2.2). Einige dieser Untermenüpunkte zeigen noch weitere Menüs, wenn sie angewählt werden.

```
Run     Compile     Edit     Options     Files     Setup     Quit

                    Editor                                    g
Line 1     Col 1     Indent   Insert   WORK.PR  Load
                                                Save
                                                Directory
                                                Print
                                                Copy
                                                Rename
                                                File Name
                                                Module list
                                                Zap file in editor
                                                Erase
                                                Operating system

                    Message                          Trace

Use first letter of option  or  select with  ->  or  <-
```

Abb. 2.2: Das Files-Menü

Falls Sie keinen weiteren Unterpunkt mehr auswählen möchten, können Sie durch Drücken der <Esc>-Taste wieder in das jeweils zuvor ausgewählte Menü zurückgehen. Wenn Sie beispielsweise jetzt die <Esc>-Taste drücken, wird

das Files-Menü verschwinden, und Sie haben nur noch den Anfangsbildschirm mit dem Hauptmenü vor sich.

Die Auswahlpunkte des Menübalkens haben folgende Funktionen:

Edit dient der Erstellung eines neuen oder der Veränderung eines bereits im Speicher befindlichen Programms.

Files lädt ein Programm von der Platte oder Diskette in den Speicher oder speichert ein momentan im Speicher befindliches Programm auf die Platte bzw. Diskette. Außerdem bietet diese Menü noch Hilfsfunktionen zur Dateiverwaltung an.

Compile compiliert ein im Speicher befindliches Programm so, daß es ohne das Prolog-System ablauffähig wird.

Run compiliert ein Programm und führt es dann aus.

Options ermöglicht die Auswahl des Dateityps für die Compilierung.

Setup verändert die Anfangsparameter (wie z. B. Fenstergrößen oder -plazierungen) und speichert, falls gewünscht, die neuen Werte für späteren Gebrauch ab.

Quit verläßt Turbo Prolog und kehrt auf die Betriebssystemebene zurück.

Erstellen eines Beispielprogramms

Wenn Sie ein Programm erstellen und ausführen wollen, können Sie nach folgenden Punkten vorgehen:

1. Wählen Sie den Edit-Modus aus, und geben Sie dann das Programm ein.
2. Verbessern Sie, wenn nötig, die Fehler.
3. Speichern Sie das Programm auf der Platte bzw. Diskette durch Auswahl der Punkte *Files* im Hauptmenü und dann *Save* im Untermenü.
4. Compilieren Sie das Programm.
5. Führen Sie es aus.

Durch den Compilierungsprozeß wird das Programm in eine sehr schnell ausführbare Form umgewandelt. Der ursprüngliche Programmtext (auch Quellcode genannt) bleibt dabei im Speicher erhalten. Sie können Ihr Programm je-

derzeit im Editor-Fenster ändern, compilieren und erneut ausführen, da bei Turbo Prolog Ihr Quellprogramm und die compilierte Version Ihres Programms gleichzeitig im Speicher vorhanden sind.

Versuchen Sie nun einmal, das einfache Prolog-Programm, das in Abb. 2.3 dargestellt ist, einzutippen, zu compilieren und auszuführen. Wählen Sie dazu zunächst den *Punkt* Edit aus.

Der Cursor wird nun auf das noch leere Editor-Fenster gesetzt. Der Cursor ist der blinkende tiefliegende Strich, der die aktuelle Schreibposition kennzeichnet. Bei der Eingabe des Programms müssen Sie am Ende jeder Zeile das Wagenrücklaufzeichen (↵) setzen, was Sie durch Drücken der Return-Taste, je nach Computertyp auch Enter- bzw. Eingabetaste genannt, erreichen. Die Eingabetaste ist bei vielen PCs als eine Taste ausgeführt, die wie ein seitenverkehrtes L aussieht und wie bei einer Schreibmaschine mit einem entsprechend abgeknickten nach links zeigenden Pfeil gekennzeichnet ist. Vergessen Sie auch nicht, die beiden Punkte einzugeben.

```
Run    Compile    Edit    Options    Files    Setup    Quit
Editor                                                  Dialog
 Line 6    Col 28    Indent  Insert  ABB2-3
/* TESTPROGRAMM */
predicates
        liebt(symbol,symbol)
clauses
        liebt(frank,susi).
        liebt(harald,ruth).
Message                                                 Trace
F1:Help F3:Search F4:Subst F5:Copy F6:Move F7:Del F8:ExtEdit F9:ExtCopy F10:End
```

Abb. 2.3: Eingabe eines Programms

Anmerkung: Bei der Texteingabe werden Sie bemerkt haben, daß der Text automatisch eingerückt wird, und zwar immer, wenn Sie auf die <Enter>-Taste drücken, nimmt der Cursor in der folgenden Zeile die Position ein, ab der in der vorigen Zeile die Texteingabe erfolgte. Es gibt auch keinen Wortumbruch. Bei einer langen Zeile wird der Text über die linke Seite des Editor-Fensters hinausgeschoben. Nur das Drücken der <Enter>-Taste bewirkt, daß eine neue Zeile beginnt. Um mit dem Cursor nach links zu gehen, können Sie die linke Pfeiltaste oder die <Rückschritttaste> verwenden, und nach rechts steuern Sie den Cursor mit der rechten <Pfeiltaste> oder der Tab-Taste. Um das automatische Einrücken (Indent) abzuschalten, müssen Sie bei niedergedrückter <Ctrl>-Taste die Tastenfolge QI eingeben.

Setzen Sie die Programmeingabe fort, auch wenn Sie einen Fehler gemacht haben. Wir werden im nächsten Abschnitt erläutern, wie Eingabefehler korrigiert werden können.

Editieren eines Programms

Sie werden sehen, wie einfach es ist, ein Programm zu editieren und die Fehler zu korrigieren. Wenn Sie schon einmal mit dem Textverarbeitungsprogramm WordStar oder mit Turbo-Pascal gearbeitet haben, werden Sie feststellen, daß viele Prolog-Befehle genauso funktionieren wie Editier-Befehle in diesen beiden Systemen. Es können Zeichen eingefügt und gelöscht werden, Blöcke können verschoben und kopiert werden, und es stehen Kommandos für Such- und Ersetzungsoperationen zur Verfügung. Eine Zusammenfassung dieser Befehle ist in der Tabelle 2.1 zu sehen. Wir wollen nun die für das Editieren wichtigsten Funktionen einzeln besprechen.

Funktion	Befehl
Cursorsteuerung	
Zeichen links	<← > oder <Ctrl> <S>
Zeichen rechts	<← > oder <Ctrl> <D>
Wort links	<Ctrl-> <←> oder <Ctrl> <A>
Wort rechts	<Ctrl-> <→> oder <Ctrl> <F>
Zeile nach oben	<↑> oder <Ctrl> <E>
Zeile nach unten	<↓ >oder <Ctrl> <X>

Funktion	Befehl
Seite nach oben	<PgUp> oder <Ctrl> <R>
Seite nach unten	<PgDn> oder <Ctrl> <C>
Zeilenanfang	<Home> oder<Ctr> <Q> <S>
Zeilenende	<End> oder <Ctrl> <Q> <D>
Dateianfang	<Ctrl> <PgUp> oder<Ctrl> <Q> <R>
Dateiende	<Ctrl> <PgDn> oder <Ctrl> <Q> <C>
Blockanfang	<Ctrl> <Q> <B>
Blockende	<Ctrl> <Q> <K>
Einfügen/Löschen	
Einfügemodus an/aus	<Ins> oder <Ctrl> <V>
Linkes Zeichen löschen	<Rückschritt>-Taste
Zeichen unter Cursor löschen	<Del> oder <Ctrl> <G>
Zeile löschen	<Ctrl> <Rückschritt> oder <Ctrl> <Y>
Bis zum Zeilenende löschen	<Ctrl> <Q> <Y>
Blockbefehle	
Blockanfang markieren	<Ctrl> <K> <B>
Blockende markieren	<Ctrl> <K> <K>
Block kopieren	<F5> oder <Ctrl> <K> <C>
Block verschieben	<F6> oder <Ctrl> <K> <V>
Block löschen	<F7> oder <Ctrl> <K> <Y>
Block von Platte lesen	<F9> oder <Ctrl> <K> <R>
Block auf Platte schreiben	<Ctrl> <K> <W>
Block-Markierung an-/ausschalten	<Ctrl> <K> <H>
Verschiedenes	
Aufruf des Hilfseditors	<F8>
Sprung auf Zeile	<F2>
Editieren beenden	<Esc> oder <F10>
Automatische Einrückung an/aus	<Ctrl> <Q> <I>
Suchen	<F3> oder <Ctrl> <Q> <F>
Weitersuchen(gleiches Kriterium)	<Shift> <F3>
Suchen und Ersetzen	<F4> oder <Ctrl> <Q> <A>

Tabelle 2.1: Editierbefehle in Turbo Prolog

Cursorsteuerung

Sie können die Pfeiltasten verwenden, um den Cursor an jeder beliebigen Stelle auf dem Bildschirm zu positionieren. Dies kann zeichenweise oder zeilenweise erfolgen, je nachdem wo Sie eine Korrektur vornehmen wollen. Die rechte Pfeiltaste bewegt den Cursor um eine Stelle nach rechts, und die linke Pfeiltaste bewegt ihn um eine Stelle nach links. Der Pfeil nach oben bringt den Cursor eine Zeile höher und der Pfeil nach unten eine Zeile tiefer.

Sie können jedoch ebenso die Cursorsteuerungsbefehle wie in WordStar benutzen:

<Ctrl> <S> – ein Zeichen nach links
<Ctrl> <D> – ein Zeichen nach rechts
<Ctrl> <E> – eine Zeile nach oben
<Ctrl> <X> – eine Zeile nach unten

Wenn Sie dabei einen Blick auf Ihre Tastatur werfen, werden Sie feststellen, daß diese vier Buchstaben ein kleines Viereck bilden, dessen Ecken jeweils der Richtung entsprechen, in die sich der Cursor bewegen soll. Dadurch kann man sich die Bedeutung dieser Funktionen leichter einprägen.

Durch Drücken der <Ctrl>-Taste zusammen mit der rechten oder linken <Pfeil>-Taste können Sie im Text wortweise vor- oder zurückspringen. Den gleichen Effekt erzielen Sie durch Verwendung der Tastenkombinationen <Ctrl> <A> für wortweises Springen nach links bzw. <Ctrl> <F> für wortweises Springen nach rechts. Wenn Sie sich diese Tasten wieder auf der Tastatur anschauen, werden Sie feststellen, daß sie gleich rechts bzw. links neben dem Viereck liegen.

Zusätzlich gibt es noch die Möglichkeit, seitenweise vor- und zurückzuspringen. Zum seitenweisen Zurückspringen dient die Taste <PgUp> oder die Kombination <Ctrl> <R> und zum seitenweisen Vorwärtsspringen die Taste <PgDn> oder<Ctrl> <C>.

Um schnell an den Anfang einer Datei oder eines Programms zu kommen, können Sie <Ctrl> <PgUp> oder <Ctrl> <Q> <R> drücken (<Ctrl> und <Q> gleichzeitig drücken, loslassen und dann <R> drücken). Um sofort an das Ende einer Datei oder eines Programms zu springen, können Sie <Ctrl> <PgDn> oder <Ctrl> <Q> <C> verwenden.

Das Drücken der <Home>-Taste oder von <Ctrl> <Q> <S> bringt Sie zum Anfang der aktuellen Zeile und <End> oder <Ctrl> <Q> <D> zum Ende.

Probieren Sie am besten die Befehle für die Cursorsteuerung an dem gerade angezeigten Programm einmal aus. Durch Verändern der Cursorposition wird nichts am Text oder am Programm geändert.

Text einfügen

Für das Einfügen von Text gibt es zwei Modi: den Einfügemodus (Insert) und den Überschreibmodus (Overwrite). Der Anwender kann mit der <Ins>-Taste oder der Tastenkombination <Ctrl> <V> zwischen den beiden Modi hin- und herschalten. Im Einfügemodus kann man den Text durch Positionieren des Cursors an der gewünschten Stelle und Eintippen des neuen Textes einfügen. Die eingegebenen Zeichen werden an der Cursorposition plaziert, und der Text, der sich rechts von diesem neuen Zeichen befindet, wird verschoben, um Platz zu schaffen. Im Überschreibmodus überschreiben oder ersetzen die eingegebenen Zeichen die Zeichen, die sich an der Cursorposition befanden, wobei kein Text verschoben wird. Standardmäßig ist bei Turbo Prolog der Einfügemodus voreingestellt.

Text löschen

Zum Löschen von Text sind zwei Tasten vorgesehen: die <Rückschritt>- und die <Del>-Taste. Die <Rückschritt>-Taste, die oben rechts auf der Tastatur durch den nach links zeigenden Pfeil gekennzeichnet ist, löscht das Zeichen, das sich links neben der aktuellen Cursorposition befindet. Daher ist diese Taste besonders zum Löschen von Zeichen geeignet, die man gerade eingetippt hat.

Die <Del>-Taste löscht das Zeichen, auf dem der Cursor sich gerade befindet. Daher verwendet man sie vorzugsweise beim Editieren. Sie müssen nur den Cursor auf das Zeichen setzen, das Sie löschen möchten, und dann die <Del>-Taste drücken.

Mit <Ctrl> <Y> können Sie die aktuelle Zeile vollständig löschen und mit <Ctrl> <Q> <Y> den Teil der Zeile, der sich rechts vom Cursor befindet.

Wenn Sie das gesamte Programm aus dem Arbeitsspeicher entfernen möchten, müssen Sie zuerst den Editor durch Drücken von <F10> oder <Esc> verlassen. Wenn es sich um ein wichtiges Programm handelt, sollten Sie es vor dem Löschen auf die Platte speichern (siehe Abschnitt "Dateioperationen: Speichern und Laden"). Nun wählen Sie den Punkt "Files" an und dort die Möglichkeit "Zap file in editor".

Das Programm erwartet vor dem Löschen von Ihnen noch eine Bestätigung. Tippen Sie "y", wenn Sie sicher sind, daß Sie das Programm im Speicher löschen wollen. Im Speicher und auf dem Bildschirm ist es nun gelöscht, aber Turbo Prolog will auch noch wissen, ob die Programmdatei auf der Platte ebenfalls gelöscht werden soll. Es zeigt bei der Eingabeaufforderung den aktuellen Dateinamen des Programms auf der Platte an. Wenn Sie nicht wollen, daß das Programm auch auf der Platte gelöscht wird, müssen Sie hier die <Esc>-Taste drücken und danach noch ein zweites Mal, um das Files-Menü wieder zu verlassen.

Anmerkung: Bei der Verwendung des Zap-Befehls müssen Sie mit Sorgfalt vorgehen, da er normalerweise sowohl zum Löschen des Programms im Speicher als auch auf der Platte verwendet wird. Wenn Sie nur das Programm im Rechnerspeicher löschen wollen, müssen Sie nach dem Löschen, wenn der Dateiname angezeigt wird, die Operation mit der <Esc>-Taste abbrechen.

Blockoperationen

Blockoperationen werden benötigt, wenn zusammenhängende Textblöcke mit einem Befehl manipuliert werden sollen. Typische Anwendungen sind das Löschen, Verschieben und Kopieren von Blöcken. Dabei sind immer zwei Schritte erforderlich: zunächst das Markieren des Blocks und dann das Ausführen der Operation.

Das Markieren eines Blocks geschieht immer in der gleichen Art und Weise. Der Cursor wird an den Anfang des Blocks gesetzt und dann <Ctrl> <K> <B> gedrückt (<Ctrl> und <K> gleichzeitig drücken, loslassen und dann <B> drükken). Dann wird der Cursor auf das Ende des Blocks gesetzt und< Ctrl> <K> <K> gedrückt.

Die Abkürzungen können Sie sich leicht merken, wenn Sie daran denken, daß mit <Ctrl> <K> die Markierung eingeleitet wird (und ebenso die meisten anderen Blockoperationen). Für den zweiten Buchstaben dient das Wort Block selbst als Mnemonik, wobei mit dem ersten Buchstaben, dem B, der Anfang des Blocks und mit dem letzten Buchstaben, dem K, das Ende markiert wird. Wenn Sie diese Operation fehlerfrei ausgeführt haben, wird der ganze Block im Vergleich zum restlichen Bildschirm in einer weniger leuchtenden Schrift dargestellt. (Wir wollen der Einfachheit halber in diesem Buch von "markierten" Blöcken sprechen.)

Ein markierter Block kann durch Drücken der Tastenkombination <Ctrl> <K> <Y> gelöscht werden. Der Block verschwindet, und der restliche Text dahinter rückt nach oben.

Wenn Sie eine Kopie des Blocks an einer anderen Stelle anfertigen wollen und der alte Block dabei an seiner Position erhalten bleiben soll, müssen Sie ihn zunächst markieren und dann den Cursor auf die neue Stelle setzen. Dann Drükken Sie <Ctrl> <K> <C> (C für Copy). Danach gibt es den Block zweimal. Sie können durch wiederholtes Drücken von <Ctrl> <K> <C> beliebig viele Kopien anfertigen.

Um einen Block zu verschieben, müssen Sie ihn ebenfalls zunächst markieren und den Cursor auf die neue Position setzen. Dann drücken Sie <Ctrl> <K> <V> (V für Verschieben). Der Block wird an der alten Stelle verschwinden und nun an der neuen eingefügt.

Wie Sie aus der Tabelle 2.1 ersehen, können für manche Blockbefehle ersatzweise auch Funktionstasten gedrückt werden. In diesem Fall brauchen Sie den Block nicht vorher zu markieren, sondern nur die Funktionstaste zu drücken und die gestellten Fragen zu beantworten.

Mit <Ctrl> <Q> <B> können Sie direkt zum Anfang des markierten Blocks springen und mit <Ctrl> <Q> <K> zum Ende. Mit der Tastenkombination <Ctrl> <K> <H> (H für Hide) kann die optische Blockanzeige aus- und wieder eingeschaltet werden. Wenn der Block nicht hervorgehoben dargestellt ist, funktionieren allerdings die anderen Blockbefehle nicht.

Um eine Kopie des Blocks als selbständige Datei auf die Platte zu schreiben, müssen Sie <Ctrl> <K> <W> verwenden (W für Write). Das Programm fragt Sie dabei nach einem Dateinamen.

Wenn Sie Blöcke von der Diskette in Ihr Programm übernehmen wollen, können Sie die Funktionstaste <F9> oder die Tastenkombination <Ctrl> <K> <R> benutzen (R für Read). Turbo Prolog bietet Ihnen die Möglichkeit, einen Teil einer gespeicherten Datei an der Stelle im Editor-Fenster einzufügen, die durch die aktuelle Cursorposition angegeben ist. Setzen Sie dazu zunächst den Cursor auf die Position, an der Sie den Text einfügen wollen. Nun drücken Sie die Funktionstaste <F9>, und das Programm bittet Sie um den Dateinamen. Nach Eingabe des Namens wird die Datei in ein neues Hilfseditor-Fenster (Aux edit) geladen.

Am unteren Rand dieses Hilfsfensters werden auch die weiteren Schritte angezeigt. Anfang und Ende des Blocks, den Sie aus dem Hilfsfenster zu übernehmen wünschen, müssen mit Hilfe der Funktionstaste <F9> markiert werden. Das Hilfsfenster verschwindet, und der neue Block erscheint an der Cursorposition im Editor-Fenster.

Das Editor-Fenster kann gelöscht werden, indem dessen gesamter Inhalt als Block markiert und gelöscht wird. Aber es ist einfacher, das Editor-Fenster zu verlassen und das Files-Menü zu verwenden.

Suchen und Ersetzen

Um eine bestimmte Zeichenkette im Programmtext zu suchen, können Sie die Funktionstaste <F3> drücken oder die Tastenkombination <Ctrl> <Q> <F> eingeben (F für Finden). Daraufhin werden Sie zur Eingabe des oder der zu suchenden Zeichen aufgefordert. Nachdem Sie erneut <F3> oder <Ctrl> <Q> <F> gedrückt haben, wird der Cursor an die Stelle bewegt, an der die Zeichenkette zum erstenmal vorkommt.

Dabei erfolgt der Suchvorgang stets vorwärts, von der aktuellen Cursorposition aus. Damit also das erste Vorkommen einer Zeichenkette gefunden werden kann, sollte der Cursor zunächst an den Programmanfang gebracht werden, bevor die Taste <F3> gedrückt wird. Um die Suche mit derselben Zeichenkette wiederholt auszuführen, brauchen Sie nur <Shift> <F3> zu drücken. Da das Wagenrücklaufzeichen in diesem Fall nicht als Endekennzeichen einer Zeichenkette dient, kann es ebenso wie alle anderen Zeichen, auch z.B. Tab, ein Bestandteil des Suchtextes sein.

Das Ersetzen einer Zeichenkette erfolgt in ähnlicher Weise, außer daß hier die Tasten <F4> oder <Ctrl> <Q> <A> zum Einsatz kommen. Diesmal müssen zwei Zeichenketten eingegeben werden: diejenige, die ersetzt werden soll, und diejenige, die ersetzen soll. Zusätzlich werden Sie gefragt, ob das Ersetzen global (alle werden ersetzt) oder lokal (nur das erste Vorkommen wird ersetzt) durchgeführt werden soll und ob Sie vor dem Ersetzen noch einmal bestätigen wollen oder nicht. Das Ersetzen kann mit Hilfe von <Shift> <F4> wiederholt ausgeführt werden.

Verschiedene Funktionen

Wenn Sie sich im Editor-Fenster befinden, können Sie durch Drücken der Funktionstaste <F8> ein Hilfseditor-Fenster aufrufen, in das Sie eine Datei laden können. Mit <F10> wird dieses Fenster wieder verlassen, und die Datei aus dem Hilfsfenster kann gespeichert werden.

Durch Drücken der Funktionstaste <F2>, Angabe einer Zeilennummer und erneutes Drücken von <F2> können Sie im Programm zu einer beliebigen Zeile springen.

Hilfsinformationen

Wenn Sie einmal nicht mehr weiter wissen, können Sie immer versuchen, den entsprechenden WordStar-Befehl einzugeben, falls Sie ihn kennen. Manchmal ist zwar noch ein zusätzlicher Tastendruck erforderlich, um das Gewünschte zu erreichen, aber man ist der Sache dann schon einen Schritt näher gekommen.

Außerdem gibt es noch die Möglichkeit, vom Editor-Fenster aus mittels der <F1>-Taste ein Fenster mit Hilfsinformationen aufzurufen. In diesem wird dann die Datei PROLOG.HLP angezeigt. Wenn Sie wollen, können Sie diese Datei nach Ihren eigenen Wünschen verändern.

Verlassen des Editors

Es gibt zwei Tasten, die man drücken kann, um das Editor-Fenster zu verlassen: die <Esc>Taste und die Funktionstaste <F10>. Meistens sind beide in ihrer Wirkungsweise identisch. Der Editor wird verlassen, und der Anwender befindet sich wieder im Hauptmenü, wo er dann einen anderen Menüpunkt auswählen kann. Der einzige Unterschied tritt auf, wenn Turbo Prolog während des Compilierens auf einen Fehler stößt. In diesem Fall kommt man automatisch in den Editor zurück. Nachdem der Fehler behoben ist, wird durch Drücken der Funktionstaste <F10> eine erneute Compilierung und Ausführung des Programms ausgelöst. Mit der Taste <Esc> würde einfach nur ein Sprung ins Hauptmenü bewirkt.

Dateioperationen: Speichern und Laden

Wenn ein Prolog-Programm erstellt wird, befindet es sich zunächst im Rechnerspeicher. Falls der Computer aus irgendeinem Grund ausfällt oder neu gestartet werden muß, geht das ganze Programm verloren, und es kann auch nicht wiederhergestellt werden. Den Rechnerspeicher bezeichnet man auch als flüchtigen Speicher, da er ohne Stromversorgung und ständige Auffrischung seine Informationen verliert und überdies bei einen Kaltstart neu initialisiert wird, damit ein definierter Ausgangszustand sichergestellt wird. Deshalb sollten Sie ein Programm von Zeit zu Zeit als Datei permanent auf Diskette oder auf der Festplatte speichern.

Das können Sie allerdings nicht vom Editor aus durchführen. Zu diesem Zweck müssen Sie den Editor vor dem Speichern des Programms mit der <Esc>-Taste verlassen. Wegen der Anfälligkeit des Rechnerspeichers gegen-

über Stromausfällen sollten Sie sich angewöhnen, etwa alle 15 bis 20 Minuten den Editor zu verlassen, das Programm zu speichern und dann in den Editor zurückzukehren. Zwingen Sie sich dazu, bis es zur Gewohnheit wird. Vielleicht werden Sie schon bald einsehen, wie sinnvoll dieser Rat ist. Ebenso sollten Sie Daten und Programme von der Festplatte zusätzlich von Zeit zu Zeit auf Disketten abspeichern, denn auch die Plattenlaufwerke bekannter PC-Anbieter sind schon häufiger ausgefallen.

Der Ablauf für das Speichern eines Programms ist der folgende:

1. Verlassen Sie den Editor mit <Esc>. Das Programm bleibt weiterhin im Editor-Fenster angezeigt.

2. Wählen Sie den Punkt "Files" im Hauptmenü durch Eingabe des Buchstabens <F> oder mit Hilfe des Cursors und anschließendem Drücken der <Enter>-Taste aus.

```
Run    Compile    Edit    Options    Files    Setup    Quit

Editor                                  Load                g
 Line 6    Col 29    Indent  Insert  ABB2-3  Save
/* TESTPROGRAMM */                         Directory
predicates                                  Print
        liebt(symbol,symbol)                Copy
clauses                                     Rename
        liebt(frank,susi).                  File Name
        liebt(harald,ruth).                 Module list
                                            Zap file in editor
                                            Erase
                                            Operating system

Message                                 Trace

Use first letter of option  or  select with  ->  or  <-
```

Abb. 2.4: Aufruf zum Speichern eines Programms

3. Im Files-Menü wählen Sie dann den Punkt "Save" durch Eingabe des

Buchstabens <S> oder mit Hilfe des Cursors und anschließendem Drükken der <Enter>-Taste aus (Abb. 2.4).

```
Run     Compile     Edit     Options     Files     Setup     Quit

Editor                                  Load                  g
 Line 6     Col 29    Indent  Insert  ABB2-3  Save
/* TESTPROGRAMM */
predicates                          File name: test
        liebt(symbol,symbol)
clauses                                 Rename
        liebt(frank,susi).              File Name
        liebt(harald,ruth).             Module list
                                        Zap file in editor
                                        Erase
                                        Operating system

Message                                 Trace

Use first letter of option  or  select with  ->  or  <-
```

Abb. 2.5: Speichern eines Programms

4. Geben Sie den Dateinamen ein, wenn Sie danach gefragt werden. Er kann bis zu 8 Zeichen lang sein. Dabei ist es nicht nötig, daß Sie das den Dateityp kennzeichnende Suffix mit angeben. Nach der Eingabe müssen Sie die <Enter>-Taste drücken (Abb. 2.5).

5. Die Datei wird unter dem angegebenen Namen mit dem Suffix .PRO zur Kennzeichnung als Programmdatei gespeichert.

6. Die Taste <Esc> bringt Sie in das Hauptmenü zurück.

7. Wenn Sie das Editieren fortsetzen wollen, können Sie wieder den Punkt "Edit" auswählen.

Ein als Datei gespeichertes Programm kann jederzeit über die Befehlsfolge "Files" und "Load" in den Editor-Bereich geladen werden. Wählen Sie dazu im Hauptmenü den Punkt "Files" und im Files-Menü die Möglichkeit "Load"

aus. Sie werden dann nach dem Dateinamen gefragt, den Sie ohne Dateityp-Suffix eingeben können. Daraufhin wird die Datei in den Arbeitsbereich geladen.

Anmerkung: Wenn eine Datei in den Arbeitsbereich geladen wird, überschreibt und zerstört sie den alten Inhalt dieses Bereichs. Deshalb sollten Sie vor dem Laden einer neuen Datei den Inhalt des Arbeitsbereichs auf Platte oder Diskette speichern, wenn Sie ihn nicht verlieren wollen.

Die anderen Auswahlpunkte des Files-Menüs haben die folgenden Bedeutungen:

Directory	gibt den Inhalt des aktuellen Verzeichnisses aus, ohne daß Turbo Prolog verlassen wird.
Print	druckt eine Kopie des Programms, das sich im Editor-Fenster befindet.
Copy	kopiert eine Datei auf der Platte.
Rename	ändert den Namen einer Datei auf der Platte.
File name	ändert den Namen der Datei im Editor-Fenster.
Module list	vergibt einen Namen für eine Liste von Modulen eines gegebenen Programmsystems; diese Option ist nützlich beim Arbeiten mit großen Programmen.
Zap file in editor	löscht den Inhalt des Editor-Fensters (beachten Sie die zuvor erwähnten Vorsichtsmaßregeln).
Erase	löscht eine Datei von der Platte.
Operating system	bringt Sie ins DOS, wo Sie einen Befehl ausführen können, wobei Turbo Prolog und Ihr Programm im Rechnerspeicher bleiben; mit *exit* kehren Sie zu Turbo Prolog zurück.

Wenn Sie noch nicht mit dem Editor von Turbo Prolog gearbeitet haben, sollten Sie jetzt einmal etwas damit üben. Korrigieren Sie zunächst das kleine Programm. Dann sollten Sie den Editor verlassen und das Programm speichern.

Kehren Sie noch einmal in den Editor zurück und probieren die Befehle aus, die Sie bis jetzt noch nicht benutzt haben. Testen Sie auch die Blockoperationen sowie das Suchen und Ersetzen. Sie sollten wirklich mit dem Editor gut vertraut sein.

Wenn Sie sich sicher genug im Umgang mit dem Editor fühlen, verlassen Sie ihn und laden die korrigierte Fassung des Programms, mit der wir nun weiterarbeiten wollen.

```
Run    Compile    Edit    Options    Files    Setup    Quit
                    Editor
 Line 1     Col 1     Indent  Insert  WORK.PRO
/* TESTPROGRAMM */                          Dialog
predicates                                 Goal:
        liebt(symbol,symbol)
clauses
        liebt(frank,susi).
        liebt(harald,ruth).
                    Message                        Trace
Compiling WORK.PRO
F8:Previous line  F9:Edit  S-F9:View windows  S-F10:Resize window  Esc:Stop exec
```

Abb. 2.6: Ausführung eines Programms

Compilieren und Ausführen eines Programms

Um ein Programm zu compilieren und auszuführen, müssen Sie den Editor durch Drücken von <F10> oder <Esc> verlassen. Wählen Sie dann den Punkt "Run" in der Hauptmenüleiste aus.

Nun wird das Programm automatisch compiliert. Im Message-Fenster werden dabei einige Meldungen erscheinen. Wenn das Programm fehlerfrei war, erscheint im Dialog-Fenster eine Eingabeaufforderung. Falls das Programm

noch Fehler enthält, kehrt Turbo Prolog zum Editor zurück. Dort können Sie noch Veränderungen vornehmen und das Programm erneut compilieren, indem Sie den Editor mit <F10> verlassen. Mit dem Wort *Goal* (Ziel) werden Sie im Dialog-Fenster aufgefordert, das gewünschte Ziel einzugeben. Versuchen Sie es beispielsweise einmal mit folgender Eingabe:

```
Goal: liebt(harald,ruth)↵
```

Die Antwort von Turbo Prolog lautet:

```
True
Goal:
```

Das bedeutet, daß Prolog mit Hilfe der Fakten, die ihm zur Verfügung standen, das Ziel, das Sie eingegeben haben, als richtig erkennen konnte.

Die Programmausführung kann jederzeit durch Drücken der <Esc>-Taste unterbrochen werden. Danach befinden Sie sich im Hauptmenü. Wenn Sie das Programm seit dem letzten Speichern verändert haben, sollten Sie es erneut speichern.

Verändern der Fensterparameter

Vielleicht haben Sie schon festgestellt, daß die voreingestellten Fenstergrößen für Ihre eigenen Zwecke nicht praktisch sind. Die meisten Beispiele in diesem Buch erfordern ein Editor-Fenster, das so groß wie der ganze Bildschirm ist. Möglicherweise wollen Sie auch das Dialog-Fenster vergrößern. Das ist kein Problem – bei Turbo Prolog ist sogar beides gleichzeitig möglich. In diesem Fall wird Turbo Prolog das jeweils aktive Fenster in den Vordergrund legen.

Beim Editieren überdeckt das Editor-Fenster, wenn es sich über den ganzen Bildschirm erstreckt, das Dialog-Fenster. Wenn dann ein Programm ausgeführt wird, überlagert das Dialog-Fenster das Editor-Fenster. Um die Fenstergrößen zu verändern, müssen Sie zunächst den Punkt "Setup" im Hauptmenü anwählen und dann die Option "Window Size".

Es erscheint ein Menü, in dem Sie das Fenster festlegen sollen, dessen Größe Sie verändern wollen. Wenn Sie ein Fenster ausgewählt haben, erscheint am unteren Rand eine Anzeige, die einen Überblick über die möglichen Funktionen zum Verschieben des Fensters und Verändern seiner Größe gibt (Abb. 2.7). Dazu werden die Pfeiltasten eingesetzt.

Wählen Sie jetzt ein Fenster aus und verschieben es oder ändern seine Größe. Dabei werden die Pfeiltasten alleine für das Ändern der Größe und Shift zusammen mit den Pfeiltasten zum Verändern der Fenster-Position verwendet.

Wenn die <Ctrl>-Taste zusammen mit der rechten und linken Pfeiltaste gedrückt wird, führt das zu einer Veränderung des Fensterformats in großen Schritten.

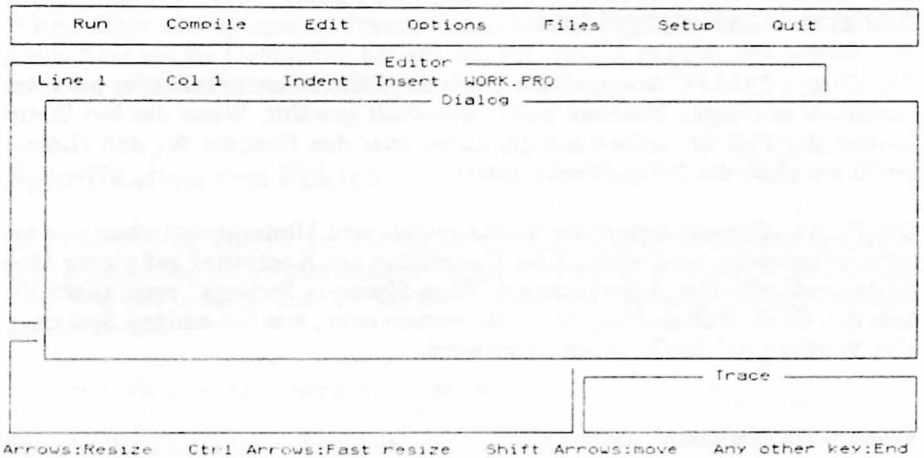


Abb. 2.7: Verändern der Fenster

Manchmal müssen Sie ein Fenster verschieben, bevor Sie seine Maße verändern können. Beispielsweise muß das Dialog-Fenster zuerst nach links verschoben werden, bevor es vergrößert werden kann, da ein Fenster sich seitlich nur nach rechts ausdehnen kann. Verändern Sie die Fenster nach Ihrem Bedarf. Mehrfaches Drücken der <Esc>-Taste bringt Sie danach ins Hauptmenü zurück. Wenn Sie wollen, können Sie die veränderten Fenstergrößen für spätere Arbeiten speichern. Dazu wählen Sie erneut die Option "Setup" und dann den Auswahlpunkt "Save". Dadurch wird die Konfiguration in der Datei PROLOG.SYS abgelegt und bei jedem Start von Turbo Prolog automatisch geladen.

Ändern des voreingestellten Plattenverzeichnisses

Turbo Prolog fragt standardmäßig das aktuelle Verzeichnis nach der Konfigurationsdatei, den Programmdateien und anderen Dateitypen ab.

Diese Standardeinstellung können Sie mit Hilfe der Auswahl "Setup" aus dem Hauptmenü für jeden der fünf Dateitypen ändern.

Wählen Sie dazu aus dem Setup-Menü den Punkt "Directories" und dort das Standardverzeichnis, das Sie ändern wollen. Diese Änderung gilt nur so lange, bis Sie Turbo Prolog verlassen, außer Sie speichern die neuen Standardverzeichnisse mit Hilfe von "Setup" und "Save".

Ändern des Hintergrunds

Bei einigen IBM PC-kompatiblen Geräten ist die Hintergrundfarbe oder die Graustufe in einigen Fenstern nicht vorteilhaft gewählt. Wenn das bei Ihrem System der Fall ist, sollten Sie die Farbe oder den Grauton für den Hintergrund mit Hilfe des Setup-Menüs ändern.

Der Punkt "Colors" ändert die Vordergrund- und Hintergrundfarben auf einem Farbmonitor, und er dient zur Einstellung des Kontrastes auf einem Monochrom-Gerät. Der Auswahlpunkt "Miscellaneous Settings" ermöglicht Ihnen, den CGA (Farbgrafikapter) zu de-/reaktivieren, was bei einigen Systemen eine Wirkung auf den Kontrast haben kann.

Wenn Schwierigkeiten auftreten

Wenn Sie sich einmal festgefahren haben und ins Hauptmenü zurückkehren wollen, dann können Sie die Tastenkombination <Ctrl> und <Break> drücken. Das bringt Sie entweder ins Hauptmenü zurück, oder Sie werden aufgefordert, die Leertaste zu drücken, was den gleichen Effekt hat.

Kapitel 3

Fakten, Objekte und Prädikate

Der größte Teil eines Prolog-Programms besteht aus einer Ansammlung von Wissen über ein bestimmtes Thema. Diese Sammlung nennt man auch *Datenbank*. Eine solche Datenbank besteht aus Fakten und Regeln. Wir wollen uns zunächst mit den Fakten beschäftigen.

Spezifikation von Fakten

In Turbo Prolog können sie Fakten durch symbolische Relationen beschreiben. Kommt beispielsweise aus dem rechten Lautsprecher Ihrer Stereo-Anlage kein Ton (ist defekt), können Sie das in deutscher Sprache folgendermaßen formulieren:

```
Der rechte Lautsprecher ist defekt.
```

Die gleiche Tatsache wird in Turbo Prolog wie folgt ausgedrückt:

```
ist(rechter_Lautsprecher,defekt).
```

Dieses Faktum wird in Prolog *clause* (Klausel) genannt. Beachten Sie dabei den Punkt am Ende der Klausel.

In dem Beispiel sind *rechter_Lautsprecher* und *defekt Objekte*. Ein Objekt ist der Name eines Elementes von einem bestimmten Typ. Es repräsentiert eine Sache oder eine Eigenschaft einer Sache aus der Realität.

Obwohl der Anwender in Turbo Prolog zwischen sechs verschiedenen Objekttypen wählen kann, wollen wir uns im Moment nur auf einen beschränken: das symbol-Objekt. Der Name eines Objekts vom Typ *symbol* beginnt in Prolog stets mit einem Kleinbuchstaben.

Die übrigen Zeichen des Namens können Großbuchstaben, Kleinbuchstaben, Ziffern oder Unterstreichungsstriche sein. Ein Name kann bis zu 250 Zeichen enthalten. Es folgen ein paar Beispiele von gültigen symbol-Objektnamen.

```
rechter_Lautsprecher
defekt
klaus
auto
rot
alter12
```

Das Wort *ist* stellte in unserem Beispiel die *Relation* oder Beziehung dar. Eine Relation ist ein Wort, das festlegt, in welcher Weise die Objekte (oder Variablen, die für Objekte stehen) miteinander verbunden sind. Dazu sehen wir uns noch ein paar Beispiele an:

Faktum	Relation
`hat_einen(theo,computer).`	`hat_einen`
`ist_ein(pudel,hund).`	`ist_ein`
`mag(nina,schokolade).`	`mag`

Der ganze Ausdruck bis zum Punkt wird *Prädikat* genannt. Ein Prädikat ist eine Funktion, die den Wert wahr oder falsch hat. Durch Prädikate wird eine Eigenschaft oder eine Beziehung ausgedrückt. Das Wort vor den Klammern ist dabei der Name der Relation. Die Elemente in den Klammern sind die *Argumente* des Prädikats, die entweder Objekte oder Variablen sein können. Im folgenden geben wir ein paar Beispiele für Prädikate:

```
angestellter(fred)

qualifiziert(maria)

familienstand(elke,verheiratet)
```

Wenn Sie hinter ein Prädikat einen Punkt setzen, erhalten Sie eine vollständige Klausel.

Bei der Bildung von Fakten können auch zusammengesetzte Prädikate verwendet werden. Beispielsweise kann der Umstand, daß Maria verheiratet und für eine Arbeit qualifiziert ist, folgendermaßen ausgedrückt werden:

```
qualifiziert(maria) and familienstand(maria,verheiratet).
```

Bis jetzt waren alle Klauseln, die wir betrachtet haben, Fakten. Es gibt noch andere Prolog-Klauseln, die sogenannten Regeln, mit denen wir uns aber später beschäftigen wollen. In Abb. 3.1 ist alles, was wir bis jetzt über Klauseln, Prädikate, Relationen und Objekte wissen, in einer Zeichnung zusammenhängend dargestellt.

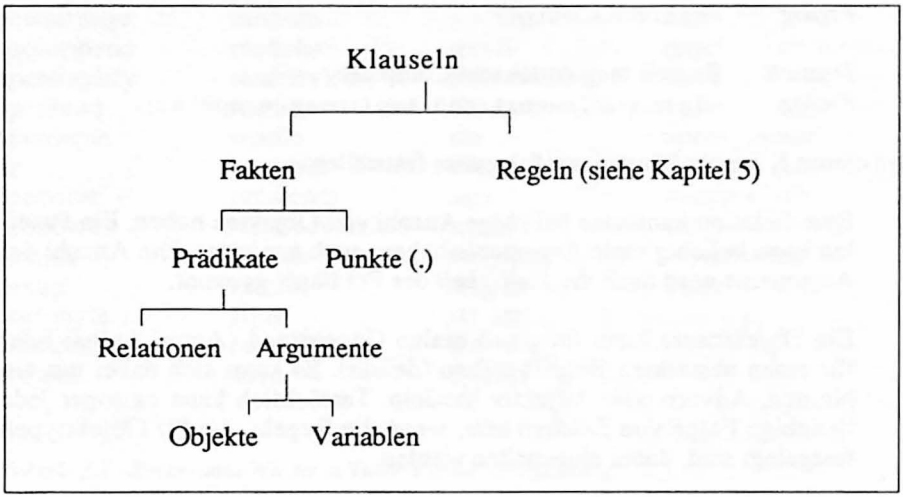


Abb. 3.1: Zusammenhang zwischen Klauseln, Prädikaten, Relationen und Objekten

Objekte in Turbo Prolog

Bevor wir fortfahren, wollen wir noch ein paar Beispiele für Fakten betrachten, die gültige Prolog-Klauseln sind:

Deutsch: Fred ist ein Angestellter.
Prolog: `angestellter(fred).`

Deutsch: Hans ist mit Maria verheiratet.
Prolog: `verheiratet_mit(hans,maria).`

Deutsch: Der Lautsprecher ist defekt.
Prolog: `diagnose(lautsprecher,defekt).`

Deutsch: Der Lautsprecher ist defekt.
Prolog: `defekter_lautsprecher.`

Deutsch: Rolf ist ein Student.
Prolog: `student(rolf).`

Deutsch: Anna ist 10 Jahre alt.
Prolog: `alter(anna,10).`

Deutsch: Roman mag Autos.
Prolog: `mag(roman,autos).`

Deutsch: Roman mag Autos und Computer.
Prolog: `mag(roman,autos) and mag(roman,computer).`

Bei diesen Klauseln können wir folgendes feststellen:

- Eine Relation kann eine beliebige Anzahl von Objekten haben. Ein Prädikat kann beliebig viele Argumente haben, auch gar keins. Die Anzahl der Argumente wird auch die *Stelligkeit* des Prädikats genannt.

- Ein Objektname kann für einen realen Gegenstand (Anna, Autos) oder für einen abstrakten Begriff stehen (defekt). Es kann sich dabei um ein Nomen, Adverb oder Adjektiv handeln. Tatsächlich kann es sogar jede beliebige Folge von Zeichen sein, wenn die Regeln, die für Objekttypen festgelegt sind, dabei eingehalten werden.

- Die reservierten Wörter von Turbo Prolog dürfen nicht zur Bildung von Objektnamen verwendet werden. Eine Liste der reservierten Wörter finden Sie in Tabelle 3.1.

abs	clearwindow	dot	global
and	closefile	edit	goal
arctan	code	editmsg	gotowindow
asserta	comline	eof	if
assertz	concat	existfile	include
attribute	config	exit	inkey
back	consult	exp	isname
beep	cos	fail	keypressed
bios	cursor	field_attr	left
bitand	cursorform	filemode	length
bitleft	database	file_str	line
bitnot	date	filepos	ln
bitor	deletefile	findall	log
bitright	diagnostics	flush	makewindow
bitxor	dir	forward	membyte
bound	disk	free	memword
char_int	display	frontchar	mod
check_determ	div	frontstr	nl
clauses	domains	fronttoken	nobreak

Tabelle 3.1: Reservierte Wörter in Turbo Prolog

not	ptr_dword	scr_attr	tan
nowarnings	random	scr_char	text
openappend	readchar	scroll	time
openmodify	readdevice	shiftwindow	trace
openread	readint	shorttrace	trail
openwrite	readln	sin	upper_lower
or	readreal	sound	window_attr
pencolor	reference	sqrt	window_str
pendown	removewindow	storage	write
penpos	renamefile	str_char	writedevice
penup	retract	str_int	writef
port_byte	right	str_len	
predicates	round	str_real	
project	save	system	

Tabelle 3.1: Reservierte Wörter in Turbo Prolog (Fortsetzung)

- Ein Prolog-Ausdruck braucht nicht alle Wörter zu verwenden, die der deutsche Satz enthält.

Sie können eine Relation definieren, wie Sie wollen, aber Sie sollten sich innerhalb eines Programms auf eine bestimmte Art festlegen. Beispielsweise kann die Relation "Hanno liebt Marga" durch

```
liebt(hanno,marga).
```

oder

```
liebt(marga,hanno).
```

ausgedrückt werden. Der zweite Ausdruck muß dabei nicht unbedingt "Marga liebt Hanno" bedeuten; es ist reine Definitionssache, wie Sie ihn interpretieren, solange Sie sich in derselben Datenbank an die einmal getroffene Vereinbarung halten.

Objekte in der Theorie der wissensbasierten Systeme

Wenn Sie bereits Bücher über Expertensysteme mit Prolog gelesen haben, kennen Sie vielleicht bereits einen anderen Objekttyp, den Sie nicht mit einem Objekt der Sprache Turbo Prolog verwechseln dürfen. Beim Entwurf eines Expertensystems wird der Fachmann von der Realität abstrahieren, um ein geistiges Ebenbild zu erschaffen, in das sowohl objektive als auch subjektive

Gesichtspunkte mit einfließen. Der Wissensverarbeitungs-Fachmann verwertet diese Abstraktion des Experten, um ein Modell zu bilden, das *Wissensdarstellung* genannt wird. Diese Wissensdarstellung ist Grundlage für ein Prolog-Programm, das dann theoretisch wie in der realen Welt reagiert (Abb. 3.2).

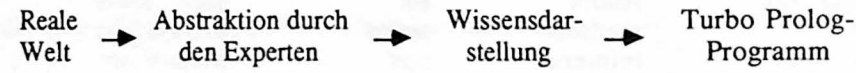


Abb. 3.2: Nachbildung der Realität durch Turbo Prolog

Bei der Wissensdarstellung werden Dinge aus der realen Welt als Objekte dargestellt. Diese Objekte haben Attribute und Eigenschaften, die wiederum mit Werten belegt sind. Beispielsweise hat eine Stereo-Anlage einen Tonkopf (Objekt); dieser Tonkopf besitzt eine Eigenschaft (den Zustand dieses Tonkopfs), die einen Wert hat (sauber oder schmutzig). Auf diese Art und Weise werden Objekte von Eigenschaften und ihren Werten unterschieden.

In Turbo Prolog ist ein Objekt etwas völlig anderes. Hier kann ein Objekt jede beliebige Konstante sein. Das Objekt in Turbo Prolog kann ein Attribut, eine Eigenschaft oder ein Wert sein. Unglücklicherweise verwendet das Handbuch zu Turbo Prolog dafür das gleiche Wort wie der Wissensverarbeitungs-Fachmann, aber solange der Anwender die beiden Begriffe auseinanderhält, sollten keine Probleme auftreten. Betrachten Sie das folgende Beispiel:

Wissensdarstellung	*Turbo Prologdarstellung*
auto → farbe → blau	ist(auto,blau).
Objekt: auto	Objekte: Auto, blau

Objekte in prozeduralen Sprachen

Wenn Sie bereits über Erfahrungen mit prozeduralen Sprachen verfügen, können Sie Prolog (eine Objekt-orientierte Sprache) einmal mit diesen Sprachen vergleichen. Grob gesagt, sind Objekte mit Konstanten in prozeduralen Programmiersprachen vergleichbar, stellen aber ein wesentlich leistungsfähigeres Konzept dar.

Bei den prozeduralen Programmiersprachen ist eine Konstante eine unveränderliche Größe – eine Zeichenkette oder ein numerischer Wert. Bei Turbo Prolog symbolisiert ein Objekt eine Sache aus der Realität.

Das Prädikat in Prolog ist mit der Funktion in anderen Sprachen vergleichbar. Wir erinnern uns, daß Prädikate Funktionen mit dem Wert wahr oder falsch sind. Die Relation entspricht einem Operator, und die Objekte sind die zugehörigen Operanden oder Funktionsargumente.

Das Turbo Prolog-Programm

Ein Programm in Turbo Prolog besteht aus mindestens zwei Abschnitten. Der Hauptteil des Programms ist der Abschnitt *clauses*, der die Klauseln enthält und aus Fakten und Regeln besteht. Klauseln, die Fakten sind, haben wir bereits kennengelernt. Im nächsten Kapitel werden wir uns mit Klauseln befassen, die aus Regeln gebildet werden.

Die Relationen, die im Abschnitt *clauses* in den Klauseln Verwendung finden, werden im Abschnitt *predicates* definiert. Jeder Relation bei den Klauseln muß die Definition eines Prädikats im Abschnitt *predicates* entsprechen. Die einzige Ausnahme bilden die eingebauten Prädikate, die fester Bestandteil von Turbo Prolog sind. Diese werden wir in Kapitel 5 kennenlernen. Die Definition eines Prädikats im Abschnitt *predicates* endet nicht mit einem Punkt. Im Abschnitt *clauses* könnte beispielsweise die folgende Klausel stehen:

```
ist(rechter_lautsprecher,defekt).
```

Dann ist es erforderlich, daß im Abschnitt *predicates* die Prädikat-Definition in der Form:

```
ist(teil,status)
```

erfolgt ist. In den Klammern könnten auch andere Namen stehen, solange gewährleistet ist, daß in der Prädikat-Definition ebenso viele Namen enthalten sind, wie das Prädikat später beim Aufruf im Abschnitt clauses erhält. Das Hauptaugenmerk bei der Definition eines Prädikats ist also auf die Angabe der entsprechenden Anzahl von Namen gerichtet, wie sie von der Relation benötigt wird; die Wörter erfüllen dabei lediglich eine Platzhalterfunktion.

Die meisten Turbo Prolog-Programme enthalten einen weiteren Abschnitt, der mit dem Wort *domains* eingeleitet wird. In diesem werden die Objekttypen festgelegt. In den meisten Programmiersprachen werden den Variablen und Konstanten (Objekte) Typbezeichnungen zugeteilt, bevor sie verwendet werden; man sagt, sie werden deklariert. Dadurch erhält der Anwender die Möglichkeit, Einfluß auf die Form der Speicherung von Konstanten und Variablen zu nehmen. Zeichenketten (wie z.B. "Hausnummer 43") werden anders

als ganze Zahlen (wie z.B. 5) abgelegt und beide wieder anders als Gleitkommazahlen (wie z.B. 5.6). In manchen Sprachen (wie C oder Pascal) wird diese Typeneinteilung in einem separaten Deklarationsteil vorgenommen. Bei anderen Sprachen wiederum ist durch den Namen der Variablen selbst schon ihr Typ dokumentiert (wie z.B. in BASIC, wo durch Anhängen von % oder $ an den Variablennamen ganze Zahlen bzw. Zeichenketten gekennzeichnet werden).

Anmerkung: In den meisten anderen Prolog-Versionen ist keine Typenvereinbarung für Variablen oder Konstanten vorgesehen. Das hat sowohl Vor- als auch Nachteile. Wenn der Benutzer ein Objekt vom falschen Typ verwendet, kann es zu einem Fehlerabbruch kommen, möglicherweise mit Verlusten von Daten auf der Festplatte oder anderen Problemen. Indem man aber in Turbo Prolog gezwungen ist, Typen vorher zu vereinbaren, entsteht eine bessere Ablaufkontrolle. Dann werden Objekte bei ihrem Auftreten überprüft, ob sie auch dem Typ entsprechen, der für sie deklariert worden ist. Wenn Sie bisher nur mit Sprachen gearbeitet haben, die keine Typenvereinbarung kannten (wie z.B. andere Prolog-Versionen oder FORTH), werden Sie diese Möglichkeit schätzen lernen.

Der Abschnitt *domains* dient also in Turbo Prolog der Typendeklaration. Dabei stehen dem Anwender sechs Grundtypen zur Verfügung: *char, integer, real, string, symbol und file* (Tabelle 3.2).

Typ	*Bedeutung*
char	Einzelnes Zeichen, in Apostrophe eingeschlossen
integer	Ganze Zahl zwischen –32768 und 32767
real	Gleitkommazahl zwischen 1E–307 und 1E+308
string	Zeichenkette, in Anführungszeichen eingeschlossen
symbol	Zeichenfolge von Buchstaben, Ziffern und Unterstrichen, wobei das erste Zeichen ein Kleinbuchstabe sein muß
file	Symbolischer Dateiname

Tabelle 3.2: Die Domain-Typen von Turbo Prolog

Im Abschnitt *clauses* könnte beispielsweise stehen:

```
ist(rechter_lautsprecher,defekt).
```

Das entsprechende Prädikat im Abschnitt *predicates* würde dann lauten:

```
ist(teil,zustand)
```

Im Abschnitt *domains* heißt es dann:

```
teil,zustand = symbol
```

Der Abschnitt *domains* kann in vielen Programmen weggelassen werden, und dem Objekt wird dann erst im Abschnitt *predicates* ein Typ zugeteilt. In dem vorangegangenen Beispiel lautet das Prädikat dann:

```
ist(symbol,symbol)
```

In diesem Fall wäre der Abschnitt *domains* überflüssig. Bei den meisten Anwendungen sollte man jedoch besser nicht so vorgehen. Es erspart dem Programmierer zwar etwas Tipparbeit, aber das Programm wird dadurch unübersichtlich, besonders wenn es viele Objekte enthält.

Die Typdeklaration für ein Objekt richtet sich nach seiner Verwendung im Programm. Wenn dieses Objekt in arithmetischen Operationen eingesetzt werden soll, sollte es vom Typ *integer* (für ganze Zahlen) oder *real* sein (für Dezimalzahlen und sehr große Zahlen).

Ganze Zahlen können natürlich als Spezialfall unter dem Typ *real* geführt werden, aber, wenn man sie als *integer* deklariert, spart man Speicherplatz und erhöht die Ausführungsgeschwindigkeit des Programms.

Der Typ *symbol* ist der allgemeinste Typ. Er kann jedoch keine Leerstellen und Kommas enthalten und muß mit einem Kleinbuchstaben beginnen. Dadurch ist seine Verwendbarkeit für Objektnamen, wie etwa Adressen, die man ausdrucken oder auf dem Schirm ausgeben will, beschränkt.

Wenn Sie einen Objektnamen drucken oder anzeigen lassen wollen, sollten Sie ihm den Typ *string* geben, wobei der Objektname dann in Anführungszeichen gesetzt werden muß, wie z. B.:

```
adresse("Heinz Berger","Schwalbenweg 11","2000 Hamburg 40")
```

Die Abschnitte in einem Turbo Prolog-Programm sollten immer in der folgenden Reihenfolge auftreten:

1. Domains (Bereiche)
2. Predicates (Prädikate)
3. Clauses (Klauseln)

Ein einfaches Turbo Prolog-Programm

Wir wollen nun sehen, wie man ein einfaches Turbo Prolog-Programm erstellt und einsetzt.

Anmerkung: Dieses Programm ist ebenso wie die anderen Programme in diesem Buch, die medizinische Diagnosen erstellen, nur als Beispiel zu verstehen. Es dient nur zur Veranschaulichung von Programmiermethoden in Turbo Prolog. Der medizinische Hintergrund ist von keinem Arzt überprüft worden.

Geben Sie das Programm aus Abb. 3.3 ein, und compilieren Sie es. Wenn nötig, können Sie den Vorgang in Kapitel 2 noch einmal nachlesen. Hier ist eine kurze Zusammenfassung der Vorgehensweise:

1. Starten Sie Turbo Prolog.
2. Wählen Sie den Edit-Modus.
3. Tippen Sie das Programm ein.
4. Verlassen Sie den Editor mit <Esc>.
5. Speichern Sie das Programm unter dem Namen TEST über Auswahl der Punkte *Files* und *Save*.
6. Wählen Sie die Option *Compile*, um das Programm zu compilieren.

Wenn ein Fehler gefunden wird, korrigieren Sie ihn und wiederholen das Vorgehen ab Schritt 4.

7. Wählen Sie *Run*, um das Programm auszuführen.

```
/* BEISPIELPROGRAMM                          */
/* NICHT FÜR DIE MEDIZINISCHE ANWENDUNG */

domains
   krankheit,anzeichen = symbol

predicates
   symptom(krankheit, anzeichen)

clauses
   symptom(windpocken,hohes_fieber).
   symptom(windpocken,schüttelfrost).
   symptom(grippe,schüttelfrost).
   symptom(erkältung,leichte_gliederschmerzen)
   symptom(erkältung,schnupfen)
   symptom(grippe,schnupfen)
   symptom(grippe,starke_gliederschmerzen)
   symptom(grippe,leichter_husten)
```

Abb. 3.3: Ein einfaches Programm

Die Klauseln in diesem Beispiel bestehen aus einer reinen Sammlung von Fakten. Wenn Sie das Programm starten, sehen Sie im Dialog-Fenster die folgende Anzeige:

```
Goal:
```

Turbo Prolog fragt nach einem Ziel. Ein Ziel ist im Prinzip eine Frage. Geben Sie einmal das folgende Ziel ein:

```
Goal: symptom(erkältung,schnupfen)↵
```

Turbo Prolog wird die Antwort *True* (wahr) geben und nach einem weiteren Ziel fragen:

```
Goal: symptom(erkältung,schnupfen)↵
True
Goal:
```

Wenn Turbo Prolog in der Datenbank ein Faktum findet, das mit der Zielaussage zu einer "Passung" gelangt, so ist die Zielaussage gelöst, und die Meldung *True* wird ausgegeben.

Diese Operation wird traditionell "Unifikation"' genannt und entspricht in der Bedeutung dem englischen Begriff "match", den Sie auch oft in der Literatur lesen können. Die Eingabe eines Ziels kann zu drei möglichen Ergebnissen führen:

1. Das Ziel wird bestätigt, d. h. der Wert *True* wird ausgegeben.
2. Das Ziel wird nicht bestätigt, d. h. Turbo Prolog konnte die Fakten des Programms nicht mit dem Ziel in Einklang bringen.
3. Die Programmausführung wird aufgrund eines Fehlers abgebrochen.

Die Ausführung eines Programms ist im Grunde genommen ein einfacher Unifikationsprozeß. Dabei beginnt Turbo Prolog mit der ersten Klausel, die ein Prädikat enthält, das mit dem Prädikat der Zielaussage unifiziert. Von diesem Punkt an prüft Turbo Prolog die Klauseln nach einer Unifikation der Argumente. Wenn eine vollständige Deckung erzielt werden konnte, gilt die Zielaussage als gelöst, und die Meldung *True* wird angezeigt.

Geben Sie einmal das folgende Ziel ein:

```
Goal: symptom(erkältung,kopfschmerzen)
```

Die Antwort von Turbo Prolog lautet:

```
Goal: symptom(erkältung,kopfschmerzen)
False
Goal:
```

Diese Antwort bedeutet nicht, daß Kopfschmerzen nicht ein Symptom für eine Erkältung sein können, sondern lediglich, daß Prolog in den Klauseln keine Unifikation für die angegebene Zielaussage finden konnte. In Prolog bedeutet die Meldung *False* also, daß keine der vorliegenden Informationen zu einer Passung geführt hat; es heißt nicht unbedingt, daß die Tatsache unrichtig ist.

Wir halten also fest, daß die Ausführung eines Prolog-Programms ein einfacher Unifikationsprozeß ist. Dabei wird keine Prozedur abgearbeitet und keine Befehlsfolge vorgeschrieben. Während der Ausführung versucht das Programm einfach nur, die angegebene Zielaussage zu einer Passung zu bringen.

Übungen

1. Finden Sie in den folgenden Beispielen die Relationen und die Argumente heraus.

```
a. adresse(hans,berlin).

b. alter(renate,14).

c. computer(ibm_pc).
```

2. Welche der folgenden Objektnamen sind für den Typ symbol gültig?

a. `rainer_neumann`

b. `Rainer Neumann`

c. `12`

d. `herbert*fischer`

e. `computer`

3. Welche der folgenden Objektnamen sind richtig, und zu welchem Typ gehören sie?

a. `"Julia Neumann"`

b. `julia neumann`

c. `julia_neumann`

d. `Alter`

e. `RolfHeinze`

f. `rolfHeinze`

g. `23.45`

h. `$43.25`

Kapitel 4

Variablen in Prolog

Bis jetzt haben wir in Prolog lediglich mit einzelnen Fakten gearbeitet, die sich bereits in einer Datenbank befanden. Wir wollen nun betrachten, wofür wir Variablen gebrauchen können.

Die Prolog-Variable

Sie können in einer Klausel oder in einer Zielaussage eine Variable verwenden, um eine unbekannte Größe zu spezifizieren. Ein Variablenname muß mit einem Großbuchstaben beginnen und kann bis zu 250 Zeichen lang sein. Mit Ausnahme des ersten Zeichens können in dem Namen beliebig Groß- und Kleinbuchstaben, Ziffern oder Unterstriche verwendet werden. Dabei sollte darauf geachtet werden, daß die Namen sinnvoll vergeben werden. Im folgenden sehen Sie ein paar Beispiele für Variablennamen:

```
Alter
Krankheit
Patient
Helmut_Berger
```

Die Datentypen in Turbo Prolog können wie in Abb. 4.1 aufgegliedert werden. Die *Objekte* aus Turbo Prolog werden in anderen Prolog-Versionen oft "atomare Sätze" genannt, und der Objekttyp *symbol* heißt dort "Atom". Listen und zusammengesetzte Strukturen werden wir in späteren Kapiteln besprechen. Die anderen Datentypen werden wir im folgenden erläutern.

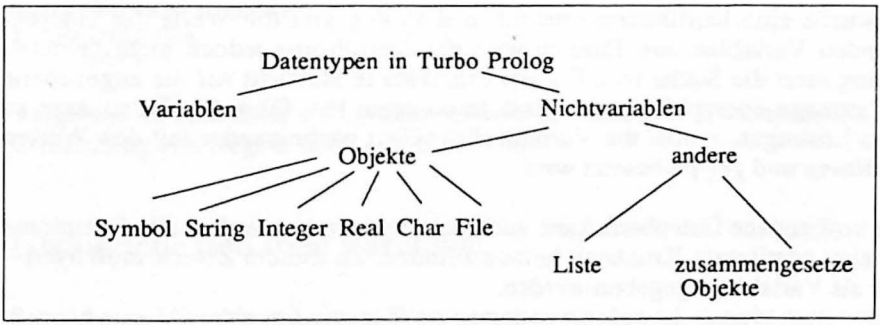


Abb. 4. 1: Datentypen in Turbo Prolog

Die Verwendung von Variablen

Laden Sie nun das Programm aus Kapitel 3 (siehe Abb. 3.3). Wir nehmen einmal an, Ihr Patient leidet unter Schnupfen, und Sie wollen eine Liste mit allen Krankheiten aus Ihrer Datenbank, für die dieses Symptom eine Rolle spielt.

Zu diesem Zweck können Sie dasselbe Programm gebrauchen und müssen nur das folgende Ziel angeben:

```
Goal: symptom(Krankheit,schnupfen)
```

Beachten sie, daß in der Zielaussage die Variable *Krankheit* angegeben ist.

Die Antwort von Turbo Prolog lautet:

```
Goal: symptom(Krankheit,schnupfen)
Krankheit=erkältung
Krankheit=grippe
2 Solutions
Goal:
```

Für das spezifizierte Symptom hat Turbo Prolog zwei Unifikationen gefunden.

Wir wollen nun untersuchen, was passiert ist, nachdem wir dieses Ziel eingegeben haben. Wie Sie auch der Abb. 4.2 entnehmen können, beginnt Turbo Prolog mit der ersten Klausel, die mit dem angegebenen Prädikat zur Passung kommt und in unserem Fall auch die erste Klausel im Programm ist.

Danach versucht es, das Objekt-Argument der Zielaussage (schnupfen) mit dem entsprechenden Argument im Klausel-Prädikat zu unifizieren. Dabei wird es nicht zu einer Passung kommen, und Prolog versucht es mit der nächsten Klausel, so lange, bis es zur Unifikation kommt.

Es wurde eine Unifikation erreicht, und Prolog gibt die Werte der entsprechenden Variablen aus. Danach wird die Ausführung jedoch nicht beendet. Prolog setzt die Suche fort, bis alle Prädikate in Hinsicht auf die angegebene Zielaussage überprüft worden sind. In unserem Fall führt die Zielaussage zu zwei Lösungen, wobei die Variable *Krankheit* nacheinander mit den Werten *erkältung* und *grippe* besetzt wird.

Die vorhandene Datenbank kann auch dazu verwendet werden, alle Symptome für eine bestimmte Krankheit herauszufinden. Zu diesem Zweck muß *Symptom* als Variable angegeben werden.

```
symptom(erkältung,schnupfen)
```

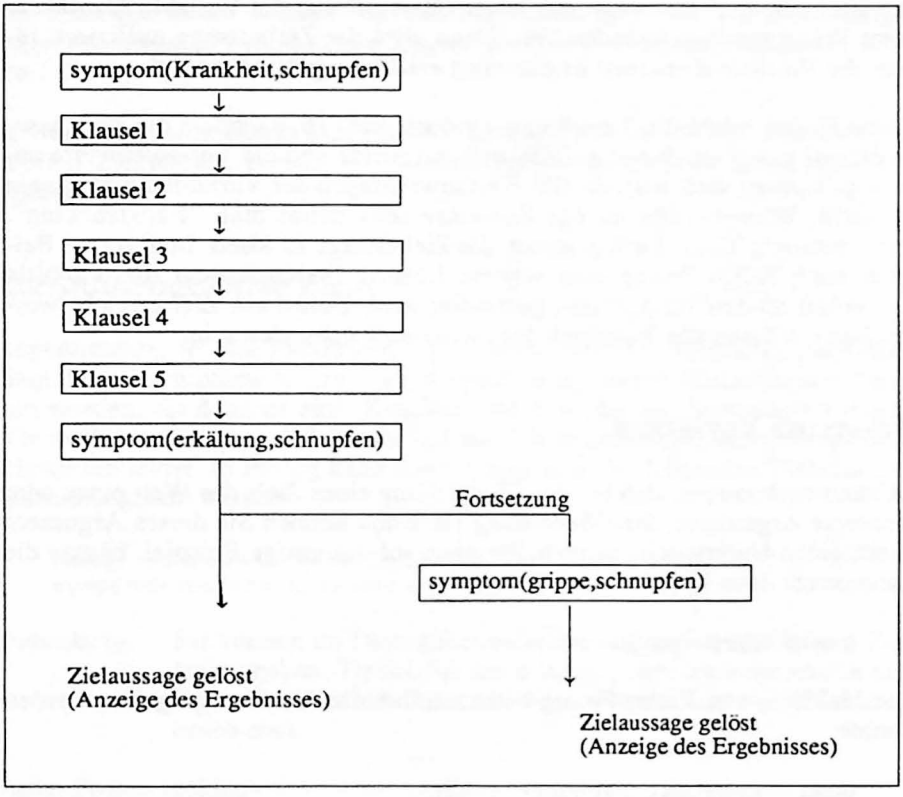


Abb. 4.2: Ablaufdiagramm eines Prolog-Programms

```
Goal: symptom(erkältung,Symptom)
Symptom=leichte_gliederschmerzen
Symptom=schnupfen
2 Solutions
   Goal:
```

Variablen können auch in Klauseln verwendet werden, wie wir später bei der Erläuterung von Regeln noch sehen werden.

Gebundene und freie Variablen

Sobald eine Variable mit einem Wert versehen worden ist, spricht man von ihr als einer *gebundenen* Variablen. Solange sie noch nicht an einen Wert gebun-

den ist, heißt sie *frei*. In unserem letzten Beispiel war die Variable *Krankheit* beim Programmstart zunächst frei. Dann wird die Zielaussage unifiziert, indem die Variable *Krankheit* an den Wert *erkältung* gebunden wird.

Turbo Prolog bricht die Verarbeitung jedoch nicht ab, nachdem die Zielaussage einmal gelöst ist. Wenn ein Ziel erreicht wurde und die Variablenwerte angezeigt worden sind, werden alle Wertzuweisungen der Variablen rückgängig gemacht. Diesen Vorgang des *Zurücksetzens* nennt man "Backtracking". Dann versucht Turbo Prolog erneut, die Zielaussage zu lösen. In unserem Beispiel wird Turbo Prolog eine weitere Lösung finden, indem die Variable *Krankheit* an den Wert *grippe* gebunden wird. Damit ein Ziel erreicht werden kann, müssen alle Variablen der Zielaussage gebunden sein.

Anonyme Variablen

Es kann vorkommen, daß bei der Überprüfung eines Ziels der Wert eines oder mehrerer Argumente ohne Bedeutung ist. Dann können Sie dieses Argument durch einen Unterstrich ersetzen. Bezogen auf das vorige Beispiel, könnte die Zielaussage dann lauten:

```
Goal: symptom(_,schüttelfrost)
```

Die Meldung von Turbo Prolog bedeutet, daß die Zielaussage gelöst werden konnte:

```
Goal: symptom(_,schüttelfrost)
True
 Goal:
```

Da keine Variable angegeben wurde, gab es auch keine Wertzuweisungen; Turbo Prolog brauchte lediglich den Namen der Relation und das letzte Argument zu überprüfen, um das Ziel zu erreichen. Wir erfahren dabei allerdings nicht, bei welcher Krankheit das Symptom *schüttelfrost* auftritt, wie es der Fall gewesen wäre, wenn wir eine normale Variable verwendet hätten. Unsere Frage lautete nur, ob es eine Krankheit mit diesem Symptom gibt, was von Turbo Prolog bejaht wurde.

Der Unterstrich kann auch in Klauseln verwendet werden, um auszudrücken, daß die Klausel für alle Argumentwerte zutreffen soll. Betrachten Sie beispielsweise die folgende Klausel:

```
mag(rosi,_).
```

Die Aussage dieser Klausel ist, daß Rosi alles mag. Für die Zielaussage *mag(rosi,computer)* würde diese Klausel dann z. B. zu einer Unifikation führen.

In einer Klausel steht die anonyme Variable also für *alle* Werte, während sie in einer Zielaussage bereits zur Lösung führt, wenn ihr mindestens ein Wert entspricht.

Zusammengesetzte Ziele

Angenommen, in Ihre Praxis käme ein Patient, der über mehrere Symptome klagt. Diese Symptome sollen in einer zusammengesetzten Zielaussage formuliert werden, die dann zu einer Krankheit führt, zu der alle Symptome passen. Wir wollen annehmen, daß der Patient unter Schnupfen und leichten Gliederschmerzen leidet. In Prolog kann diese Tatsache in der folgenden Zielaussage zusammengefaßt werden:

```
Goal: symptom(Krankheit,schnupfen) and
symptom(Krankheit,leichte_gliederschmerzen)
```

Anmerkung: Sie können im Dialog-Fenster ohne weiteres solche langen Zeilen eingeben. Tippen Sie nur alles ein, ohne zwischendurch die Enter-Taste zu drücken; es findet automatisch ein Zeilenumbruch statt.

Turbo Prolog meldet:

```
Goal: symptom(Krankheit,schnupfen) and
symptom(Krankheit,leichte_gliederschmerzen)
Krankheit=erkältung
1 Solution
Goal:
```

Wir sprechen hier von einem zusammengesetzten Ziel, was bedeutet, daß alle angegebenen Bedingungen erfüllt sein müssen, damit das Ziel erreicht wird. Bei der Überprüfung des zusammengesetzten Ziels geht Prolog von links nach rechts vor, so wie man liest. Sobald der erste Teil des Ziels zu einer Passung gelangt, ist die Variable *Krankheit* gebunden. Falls die gleiche Variable noch in anderen Termen der Zielaussage vorkommt, muß sie auch dort mit dem gleichen Wert passen.

Wir wollen nun betrachten, wie im vorliegenden Fall bei der Überprüfung des Ziels vorgegangen wird. Zu Beginn der Suche ist die Variable *Krankheit* noch

frei. Sobald die Klausel *symptom(erkältung,schnupfen)* gefunden wird, wird die Variable *Krankheit* an den Wert *erkältung* gebunden. Damit ist der erste Teil (Term) der Zielaussage gelöst, und Prolog versucht nun, den zweiten Teil zu lösen. Die Variable *Krankheit* ist nun gebunden, und Prolog markiert diese Stelle in der Liste der Klauseln, um später darauf zurückzukommen. Wie Sie anhand der Abb. 4.3 nachvollziehen können, versucht Prolog nun, die zweite Teilaussage zu überprüfen, indem es wieder mit der ersten Klausel beginnt, die mit dieser Teilaussage unifiziert. Da die Variable *Krankheit* nun mit dem Wert *erkältung* besetzt ist, gilt dieser Wert auch für den zweiten Teil der Zielaussage. Es kommt zu einer Passung mit der Klausel *symptom(erkältung,leichte_gliederschmerzen)*. Da nun beide Teile des zusammengesetzten Ziels zum Erfolg führten, gilt die Zielaussage als gelöst, und das Ergebnis wird angezeigt.

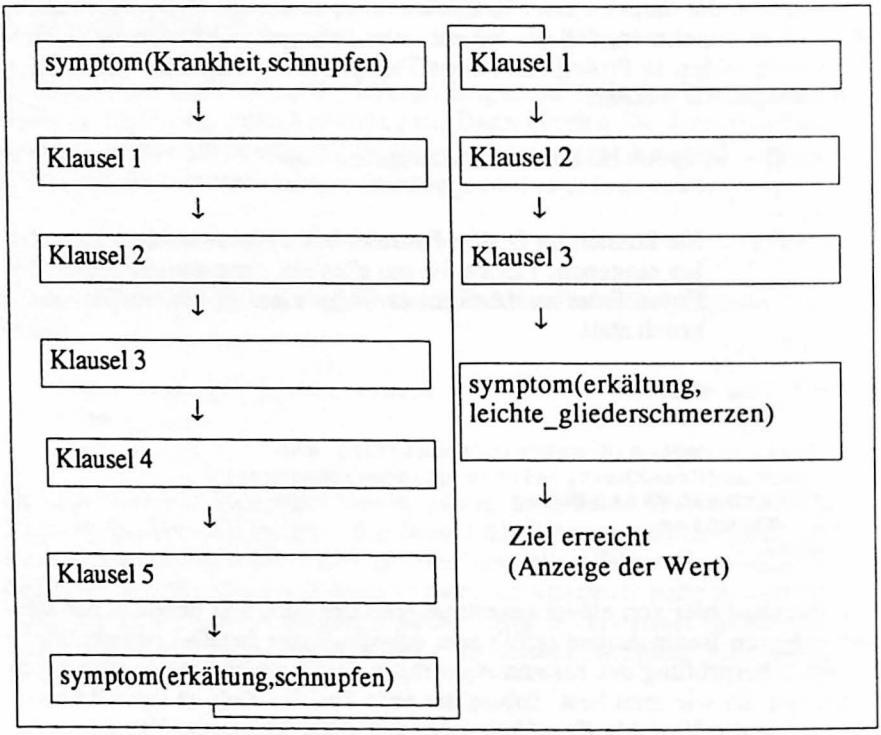


Abb. 4.3: Programmablauf bei einem zusammengesetzten Ziel

Danach wird die Variable *Krankheit* von ihrem Wert gelöst, Turbo Prolog kehrt an die markierte Stelle zurück, an der die Wertzuweisung für die Va-

riable erfolgt war, und sucht ab da nach einer weiteren Passung. Diese erfolgt mit der Klausel *symptom(grippe,schnupfen)*. Dieses Mal erhält die Variable *Krankheit* also den Wert *grippe*, und Prolog überprüft nun wieder den zweiten Teil der zusammengesetzten Zielaussage, wobei *grippe* der Wert für *Krankheit* ist.

Die Klausel *symptom(grippe,schüttelfrost)* wird gefunden, führt aber nicht zum Ziel. Die Variable *Krankheit* wird erneut freigegeben, und die Suche wird ab der letzten markierten Stelle fortgesetzt, an der die Variable *Krankheit* den Wert *grippe* erhielt. Die Zielaussage kann nicht unifiziert werden, und die Ausführung wird beendet.

Backtracking

Bei der Lösung des vorigen Beispiels wurde eine der wichtigsten Techniken bei der Programmausführung von Prolog vorgestellt: das *Backtracking*. Die Überprüfung eines zusammengesetzten Ziels erfolgt von links nach rechts.

Wenn irgendeine der Bedingungen nicht erfüllt werden kann, geht Prolog zur vorigen Bedingung zurück, um einen erneuten Versuch mit einem anderen Variablenwert zu starten, der möglicherweise die Bedingung erfüllt. Dabei geht Prolog unablässig vor und zurück und versucht dabei, mit jedem verfügbaren Wert alle möglichen Lösungen zu finden.

Obwohl Prolog eine Objekt-orientierte Sprache ist, kann man doch den Weg, den es zur Lösung einer Zielaussage einschlägt, ziemlich genau vorhersagen. Prolog kann aber insofern als Objekt-orientiert bezeichnet werden, als der Weg (Prozedur), der zur Lösung der Zielaussage führt, über eine Datenbank (Fakten und Regeln über Objekte und ihre Beziehungen) ermittelt wird.

Der Programmierer braucht dabei keine Vorgehensweise anzugeben, da dies eine interne Aufgabe von Prolog ist und ohne weitere Maßnahmen automatisch erfolgt.

Bevor Sie fortfahren, sollten Sie noch ein paar Beispiele für zusammengesetzte Ziele an unserem Diagnose-Programm ausprobieren (siehe auch Übung 2).

Machen Sie sich das Prinzip des Backtrackings an diesem einfachen Beispiel klar, bevor Sie zu den schwierigeren Beispielen des nächsten Kapitels übergehen.

Vereinbarungen für Variablen

Für Variablen in Turbo Prolog gelten die folgenden vier Grundregeln:

1. Variablennamen müssen mit einem Großbuchstaben beginnen. Sie können bis zu 250 Zeichen lang sein und mit Ausnahme des ersten Zeichens beliebig aus Klein- und Großbuchstaben, Ziffern und Unterstrichen zusammengesetzt sein.

2. Zu einer bestimmten Zeit ist eine Variable entweder gebunden oder frei.

3. Die Festlegung einer Variablen ist immer an eine bestimmte Klausel gebunden. Eine Variable mit dem gleichen Namen in einer anderen Klausel wird wie eine andere Variable behandelt. In einfachen und zusammengesetzten Zielaussagen gilt der gleiche Wert für jedes Auftreten des gleichen Variablennamens.

4. Sobald eine Variable an ein Objekt gebunden ist, erhält sie den gleichen Typ wie dieses. Wenn beispielsweise das Argument *alter* im Abschnitt *predicates* oder *domains* als ganze Zahl deklariert worden ist und die Variable *Angestellter_Alter* in einer der Klauseln an dieses Objekt gebunden wird, so wird *Angestellter_Alter* danach wie eine ganze Zahl behandelt. Es wird dann auch zu keiner Unifikation mit einem Argument eines anderen Typs in einer anderen Klausel kommen.

Abschließende Bemerkungen

An jeder beliebigen Stelle eines Prolog-Programms kann ein Kommentar eingefügt werden. Gute Programmierung zeichnet sich dadurch aus, daß ausreichend viele Kommentare in ein Programm eingefügt werden, um dem Anwender das Verständnis zu erleichtern. Auf jeden Fall sollte jedes Programm mit einer Überschrift und dem Datum versehen werden.

Ein Kommentar wird durch die Zeichenfolge /* eingeleitet und endet mit */. Man kann die Kommentare in Zeilen setzen, die bereits Befehle enthalten oder auch in separate Zeilen.

```
/* MEDIZINISCHE DIAGNOSE          */
/* APRIL 1987                     */
/* BEISPIELPROGRAMM - NICHT       */
/* FÜR PROFESSIONELLEN            */
/* GEBRAUCH GEEIGNET              */
```

```
domains                  /* Bereiche */
    person1,person2 = symbol

predicates             /* Prädikate */
   mag(person1,person2)

clauses                  /* Klauseln */
    mag(maria,hans).  /* Beispiel */
```

Übungen

1. Angenommen, wir haben die folgende Datenbank erstellt:

```
ort(essen,brd)
ort(paris,frankreich)
ort(florenz,italien)
ort(rom,italien)
ort(tutzing,brd)
```

a. Formulieren Sie eine Zielaussage, die zur Ausgabe aller deutschen Städte (brd) führt.

b. Formulieren Sie eine Zielaussage, um herauszufinden, in welchem Land die Stadt Florenz liegt.

2. Zu welchen Ergebnissen führen die folgenden Zielaussagen, wenn Sie die Datenbank aus dem Beispiel der Abb. 3.3 zugrundelegen?

a. **Goal:** `symptom(X,schnupfen) and symptom(X,leichte_gliederschmerzen)`

b. **Goal:** `symptom(X,schnupfen) and symptom(Y,leichte_gliederschmerzen)`

c. **Goal:** `symptom(X,schnupfen) and symptom(X,Y)`

d. **Goal:** `symptom(erkältung,X) and symptom(grippe,X)`

Kapitel 5

Verwendung von Regeln

Wir wissen nun bereits, wie man mit Prolog einfache medizinische Diagnosen stellen kann. Mit einer kleinen Datenbank aus Fakten ist ein produktives und nützliches System für medizinische Diagnosen entstanden. Wir wollen nun einen Schritt weitergehen und die Datenbank mit Regeln erweitern. Mit Hilfe von Regeln kann Prolog aus bereits bestehenden Fakten neue Fakten ableiten.

Regeln

Eine *Regel* ist ein Ausdruck, der bewirkt, daß der Wahrheitswert eines bestimmten Faktums von einem oder mehreren anderen Fakten abhängig ist. Betrachten Sie das folgende Beispiel:

IF (Wenn) Gelenkschmerzen auftreten

AND (und) eine Anfälligkeit für Infektionen vorliegt,

THEN (dann) könnte ein Mangel an Vitamin C vorliegen.

Diese Regel kann in Turbo Prolog folgendermaßen ausgedrückt werden:

```
annahme(vitc_mangel) if
symptom(arthritis) and
symptom(infektions_anfälligkeit).
```

Beachten Sie die allgemeine Form einer Prolog-Regel. Die Schlußfolgerung wird zuerst angegeben, und danach folgt das Wort *if*. Als nächstes werden die Bedingungen, von denen die Schlußfolgerung abhängt, aufgeführt, wobei zwischen zwei Bedingungen das Wort *and* gesetzt wird. Die Regel wird ebenso wie ein Faktum durch einen Punkt abgeschlossen. Die Schlußfolgerung ist im Grunde genommen nichts anderes als ein Prolog-Ziel. Dieses Ziel wird erreicht (True), wenn alle für das Ziel angegebenen Bedingungen zutreffen.

Jede Regel besteht aus einem linken Teil, dem Regelkopf, und einem rechten Teil, dem Regelrumpf. Der Regelkopf besteht aus einem einzigen Term, wäh-

rend der Regelrumpf eine Folge von Termen ist. Die Terme des Regelrumpfes bilden eine Konjunktion (Verknüpfung) von Zielen, die alle erfüllt sein müssen, damit der Regelkopf wahr ist. Wenn alle Bedingungen zutreffen, ist das Ziel erreicht; wenn jedoch nur eine Bedingung nicht erfüllt ist, trifft die ganze Schlußfolgerung nicht zu.

Eine Regel gibt eine Beziehung zwischen Fakten an. Ein Prolog-Programm ist eigentlich nichts anderes als eine einfache Datenbank (Sammlung von Klauseln) mit Fakten und Regeln.

Beachten Sie, daß der Regelkopf und jede Bedingung in eine eigene Zeile geschrieben wurden. Diese Schreibweise dient nur der besseren Lesbarkeit; für eine korrekte Ausführung des Programms ist sie nicht erforderlich. Bei der Annahme und den einzelnen Bedingungen handelt es sich um Prädikate, die den Wert wahr oder falsch haben können.

Da Regeln einen wesentlichen Teil eines jeden Prolog-Programms ausmachen, hat man eine verkürzte Schreibweise dafür entwickelt. Die oben formulierte Regel lautet in dieser verkürzten Form:

```
annahme(vitc_mangel):-
symptom(arthritis),
symptom(infektions_anfälligkeit).
```

Das Komma repräsentiert dabei eine UND-Verknüpfung, während ein Semikolon für eine ODER-Verknüpfung verwendet werden kann. Wir empfehlen aber der Übersichtlichkeit halber, eine ODER-Verknüpfung durch zwei Regeln auszudrücken:

```
annahme(vitc_mangel):-
     symptom(arthritis),
     symptom(infektions_anfälligkeit).
annahme(vitc_mangel):-
     symptom(kolitis),
     symptom(infektions_anfälligkeit).
```

Umgangssprachlich ausgedrückt, bedeuten diese Ausdrücke, daß ein Mangel an Vitamin C besteht, wenn entweder eine Arthritis *und* eine Infektionsanfälligkeit vorliegt *oder* eine Kolitis *und* eine Infektionsanfälligkeit. Wenn eins der beiden Bedingungspaare erfüllt ist, so ist die Schlußfolgerung wahr.

Einsatz von Regeln bei Problemlösungen

Die Vorgehensweise, bei der Regeln und Fakten zur Lösung eines Problems verwendet werden, nennt man *formale Schlußfolgerung*. Jeder von uns ist bei

der Lösung seiner alltäglichen Probleme schon auf diese Art und Weise vorgegangen. Ein klassisches Beispiel für formale Schlußfolgerung finden Sie in Krimirätseln, bei denen sich der Detektiv durch ein Labyrinth von Fakten durcharbeiten muß und das Rätsel aufgrund formaler Schlußfolgerungen lösen kann.

Dabei soll der Leser mit Hilfe von Indizien den Fall möglichst schneller als der Detektiv aufklären. Verdächtige mit stichhaltigen Alibis werden ausgeschlossen. Es werden Motive überprüft, um herauszufinden, aus welchen Gründen die Verdächtigen das Verbrechen begangen haben könnten. Auch die Art und Weise, wie das Opfer getötet wurde, läßt Schlüsse zu. Laboruntersuchungen liefern meistens zusätzliche Hinweise. Schließlich zieht der Detektiv mit Hilfe der Fakten seine Schlußfolgerungen.

Schritt für Schritt wird der Detektiv aufgrund der Beziehungen zwischen den Fakten, die in Regeln ausgedrückt werden, zu Zwischenergebnissen gelangen und damit immer näher an die Lösung des Endziels kommen. Das könnte etwa so aussehen:

IF (Wenn) der Verdächtige um 22.15 Uhr in der Lulu-Bar gesehen wurde

AND (und) das Opfer um 22.30 Uhr zu Hause getötet wurde

AND (und) die Lulu-Bar vom Haus des Opfers 30 Minuten entfernt liegt,

THEN (dann) kann der Verdächtige nicht der Täter sein.

Wie Sie sehen, können Sie mit Hilfe eines Prolog-Programms Indizien überprüfen und ein Krimirätsel lösen.

Verwendung von Variablen in Regeln

Variablen können in Regeln dazu verwendet werden, eine generelle Aussage über eine Beziehung zu machen. Betrachten Sie beispielsweise die folgende Regel:

```
annahme(Patient,masern):-
  symptom(Patient,fieber),
  symptom(Patient,husten),
  symptom(Patient,bindehautentzündung),
  symptom(Patient,schnupfen),
  symptom(Patient,hautausschlag).
```

Die Regel besagt, daß ein Patient, der unter Fieber, Husten, Bindehautentzündung, Schnupfen und Hautausschlag leidet, zweifellos Masern hat.

Wir wollen nun ein zusammenhängendes Beispiel betrachten. Tippen Sie das kurze Programm aus Abb. 5.1 ein, und compilieren Sie es. Dieses Programm kann für die Diagnose von Kinderkrankheiten eingesetzt werden.

```
/* MEDIZINISCHES DIAGNOSE-SYSTEM */
/* FÜR KINDERKRANKHEITEN         */
/* APRIL 1987                    */
/* BEISPIELPROGRAMM - NICHT FÜR  */
/* PROFESSIONELLEN GEBRAUCH      */
domains
    krankheit,indikation,name = symbol

predicates
    annahme(name,krankheit)
    symptom(name,indikation)

clauses
    symptom(rolf,fieber).

    symptom(rolf,hautausschlag).

    symptom(rolf,kopfschmerzen).

    symptom(rolf,laufende_nase).

    annahme(Patient,masern):-
        symptom(Patient,fieber),
        symptom(Patient,husten),
        symptom(Patient,bindehautentzündung),
        symptom(Patient,laufende_nase),
        symptom(Patient,hautausschlag).

    annahme(Patient,röteln):-
        symptom(Patient,fieber),
        symptom(Patient,kopfschmerzen),
        symptom(Patient,laufende_nase),
        symptom(Patient,hautausschlag).

    annahme(Patient,grippe):-
        symptom(Patient,fieber),
        symptom(Patient,kopfschmerzen),
        symptom(Patient,gliederschmerzen),
        symptom(Patient,bindehautentzündung),
        symptom(Patient,schüttelfrost),
        symptom(Patient,halsschmerzen),
        symptom(Patient,husten),
        symptom(Patient,laufende_nase).

    annahme(Patient,erkältung):-
```

```
symptom(Patient,kopfschmerzen),
    symptom(Patient,niesen),
    symptom(Patient,halsschmerzen),
    symptom(Patient,schüttelfrost),
    symptom(Patient,laufende_nase).

annahme(Patient,mumps):-
    symptom(Patient,fieber),
    symptom(Patient,geschwollene_drüsen).

annahme(Patient,windpocken):-
    symptom(Patient,fieber),
    symptom(Patient,hautausschlag),
    symptom(Patient,gliederschmerzen),
    symptom(Patient,schüttelfrost).

annahme(Patient,keuchhusten):-
    symptom(Patient,husten),
    symptom(Patient,niesen),
    symptom(Patient,laufende_nase).
```

Abb. 5.1: Ein medizinisches Diagnose-System für Kinderkrankheiten

Durch einen Blick auf unsere Datenbank können wir feststellen, daß Rolf krank ist. Seine Symptome (Fieber, Hautausschlag, Kopfschmerzen und Schnupfen) sind als Fakten in die Datenbank aufgenommen worden. Hinzu kommen sieben Regeln, mit deren Hilfe Beziehungen zwischen den bekannten Fakten hergestellt werden können. Wenn das Programm eine Eingabe verlangt, sollten Sie das folgende eintippen:

```
Goal: annahme(Patient,Krankheit)
```

Das ist eine Aufforderung an Prolog, alle Patienten und ihre entsprechenden Krankheiten aufzulisten. Das Programm antwortet:

```
Goal: annahme(Patient,Krankheit)
Patient=rolf, Krankheit=röteln
1 Solution    Goal:
```

Zwei Punkte sind zu beachten:

- Der Variablenname der Zielaussage muß nicht mit dem in der Klausel übereinstimmen; wichtig ist nur die Position innerhalb des Prädikats. Variablenwerte werden durch den Unifikationsprozeß übergeben, worauf wir später in diesem Kapitel noch genauer zu sprechen kommen.

- Die Namen der Argumente im Abschnitt *predicates* sind einfach nur Platzhalter. Sie sind von den Namen in den Klauseln oder Zielaussagen unabhängig, müssen aber im Abschnitt *domains* einem Typ zugeordnet werden.

Wir wollen nun nachvollziehen, was abläuft, nachdem wir das Ziel eingegeben haben. Prolog geht sofort zur ersten Regel, deren Kopf mit der Zielaussage oder ihrem ersten Term unifiziert werden kann:

```
annahme(Patient,masern)
```

Prolog markiert diese Stelle, betrachtet diesen Regelkopf als Zielaussage und versucht sie zu lösen. Dabei muß zunächst die erste Bedingung überprüft werden:

```
symptom(Patient,fieber)
```

Nun beginnt Prolog beim Kopf der ersten Regel oder des Faktums, die oder das mit diesem Prädikat zur Passung gelangt. Das ist in diesem Fall die erste Klausel des Programms:

```
symptom(rolf,fieber)
```

Die Passung gelingt, und die Variable *Patient* wird an den Wert *rolf* gebunden. Dieser Variablenwert wird an die ursprüngliche Regel übergeben.

Dann geht Prolog zur nächsten Bedingung in der Regel weiter, wobei die bereits erfolgte Wertzuweisung *rolf* bestehen bleibt:

```
symptom(rolf,husten)
```

Hierbei kommt es nicht zu einer Passung, was das Scheitern des Regelkopfs *annahme(rolf,masern)* mit sich bringt. Danach geht Prolog zurück und versucht, eine andere Unifikation für die ursprüngliche Zielaussage zu finden. Das Ziel *annahme(Patient,Krankheit)* wird als nächstes mit

```
annahme(Patient,röteln)
```

unifiziert. Hier ist die Variable *Patient* nicht die gleiche Variable wie in der ersten Klausel. Es wird auch kein Variablenwert von der ersten Klausel an diese übergeben. In jeder Klausel können die Variablen erst durch den Unifikationsprozeß gebunden werden.

Wieder wird zuerst die erste Bedingung überprüft und die Variable *Patient* wieder an den Wert *rolf* gebunden. Die vier weiteren Voraussetzungen für eine Diagnose der Krankheit Röteln sind ebenfalls alle erfüllt, was dazu führt, daß das angegebene Ziel mit *annahme(rolf,röteln)* erreicht wird. Die Variablen *Patient* und *Krankheit* der ursprünglichen Zielaussage sind nun beide gebunden und werden angezeigt.

Prolog probiert aber noch weiter, die angegebene Zielaussage mit verschiedenen Wertzuweisungen zu lösen. Im vorliegenden Programm wird die ursprüngliche Zielaussage mit jedem Regelkopf zur Passung gelangen, und deshalb wird auch jede Regel überprüft, bevor das Progamm zum Ende kommt. In jedem Fall wird aber mindestens eine Bedingung nicht erfüllt sein, so daß es keine weitere Lösung gibt.

Programmausführung in Prolog

Wir können das bisher Gesagte folgendermaßen zusammenfassen:

- Die Programmausführung in Prolog ist ein *Unifikationsprozeß*.

- Für jedes angegebene Ziel versucht Prolog, ein Faktum oder einen Regelkopf zu finden, das oder der mit dieser Zielaussage zu einer *Passung* gelangt oder *unifiziert* (manchmal auch dem Englischen entsprechend *matched* genannt).

- Wenn ein entsprechendes Faktum gefunden wird, ist die Zielaussage sofort gelöst.

- Wenn eine Regel gefunden wird, versucht Prolog, den Kopf dieser Regel zu beweisen, indem es den Regelrumpf als neues zusammengesetztes Ziel betrachtet und nun jede Bedingung dieses Rumpfes überprüft. Wenn eine der Bedingungen nicht erfüllt ist, geht Prolog zurück (*Backtracking*) und beginnt wieder damit, die vorausgegangenen Bedingungen mit anderen Werten in Einklang zu bringen.

  Wenn das auch nicht zum Ziel führt, wird nach einem anderen Faktum oder einer anderen Regel gesucht, die mit dem ursprünglichen Ziel unifiziert. Nur wenn alle Bedingungen des Regelrumpfes erfüllt sind, ist die Zielaussage gelöst.

- Die Programmausführung in Turbo Prolog endet so lange nicht, bis daß alle möglichen Lösungen für das Ziel überprüft wurden.

Beachten Sie auch die Verwendung der Variablen. Es gibt keine *globalen* Variablen, d. h. es gibt keine Variablen, die ihren Wert während der ganzen Programmausführung behalten. Alle Variablen sind *lokal* in bezug auf eine Klausel, in der sie auftreten. Sogar, wenn der gleiche Variablenname in einer anderen Klausel verwendet wird, handelt es sich nicht um die gleiche Variable.

Wenn die Überprüfung einer Klausel nicht zum Erfolg führt, findet das sogenannte *Backtracking* statt. Das bedeutet, daß alle gebundenen Variablenwerte wieder freigegeben werden und ein neuer Versuch ab der markierten Stelle gestartet wird, an der die Variablen gebunden wurden, um eine Lösung der Zielaussage mit anderen Werten zu erreichen.

In unserem letzten Beispiel war die Variable *Patient* die einzige in allen Regeln. Werte werden an Fakten und Regeln nach den Gesetzen der Unifikation übergeben (siehe Abschnitt "Unifikation").

Verwendung eines Ablaufprotokolls

Um eine Programmausführung Schritt für Schritt nachvollziehen zu können, gibt es den Trace-Modus. Schreiben Sie als erstes das Wort *trace* in Ihr Programm, noch vor dem Abschnitt domains.

```
trace
domains
```

Compilieren Sie das Programm, und geben Sie dann das folgende Ziel ein:

```
Goal: annahme(Patient,Krankheit)
```

Das Programm wird nach jedem Schritt die Ausführung unterbrechen und im Trace-Fenster eine Meldung ausgeben. Am unteren Bildschirmrand erscheint eine Menüzeile mit Bedienerhinweisen. Die Funktionstaste <F10> führt Sie in Einzelschritten durch das Programm. Wenn Sie unser Beispielprogramm einmal auf diese Weise ausführen lassen, werden Sie sehen, wie der Programmablauf den Beschreibungen dieses Kapitels entspricht. Um die Verarbeitung abzubrechen, müssen Sie die <Esc>-Taste drücken.

Unifikation

In Prolog ist ein *Term* ein einfaches Objekt, eine Variable oder eine Struktur, wie etwa eine Liste oder zusammengesetzte Struktur (darüber erfahren Sie

später noch mehr). Der Vorgang, bei dem Prolog eine Übereinstimmung eines Terms mit Fakten oder Köpfen von anderen Regel sucht, um ein Ziel zu erreichen, nennt man *Unifikation*. Eine Unifikation ist ein Mustererkennender Prozeß.

Ein Term *unifiziert* mit einem anderen Term, wenn

- beide Terme in Prädikaten mit der gleichen Anzahl von Argumenten vorkommen und dort auch an der gleichen Position stehen
- beide Terme vom gleichen Typ sind - der Typ symbol kann nur mit dem Typ symbol unifizieren usw
- alle Subterme auch miteinander unifizieren. (Wir werden später noch sehen, daß ein Term auch ein zusammengesetztes Objekt oder eine Liste sein kann. In diesem Fall müssen auch alle Subterme unifizieren.)

Für die Unifikation gelten folgende Aussagen:

- Ein freie Variable unifiziert mit jedem Term, der die vorangegangenen Bedingungen erfüllt. Nach der Unifikation ist die Variable an den Wert des Terms gebunden.
- Ein Konstante unifiziert mit sich selbst oder mit einer freien Variablen.Wenn sie mit einer Variablen unifiziert, wird diese an den Wert der Konstanten gebunden.
- Eine freie Variable unifiziert mit jeder anderen freien Variablen. Nach der Unifikation werden die beiden Variablen wie eine einzige behandelt. Wenn eine der beiden Variablen gebunden wird, so wird auch die andere an den gleichen Wert gebunden.

Prädikate unifizieren miteinander, wenn

- sie den gleichen Relationsnamen haben
- sie die gleiche Anzahl von Argumenten besitzen
- alle Argumentepaare miteinander unifizieren

In unserem Programm kam es beispielsweise zur Unifikation von *annahme(Patient,Krankheit)* mit *annahme(rolf,röteln)*. Dabei wurde die Variable *Patient* an den Wert *rolf* und die Variable *Krankheit* an den wert *röteln* gebunden.

Die Unifikation kann man mit der Parameterübergabe in prozeduralen Programmen vergleichen. Dabei werden die Werte eines Terms an einen anderen Term übergeben, wobei alle Variablen in diesem Term gebunden werden.

Steuerung des Programmablaufs

Auch wenn Prolog keine prozedurale Sprache ist, so kann doch eine gewisse Kontrolle über die Programmausführung ausgeübt werden. Dabei befolgt Prolog ein paar allgemeine Regeln:

- In einem Programm müssen alle Klauseln mit dem gleichen Prädikat in einer Gruppe zusammenstehen. In unserem Beispiel sind alle Regeln mit dem Regelkopf *annahme(Patient,Krankheit)* zusammengefaßt worden. In dieser Gruppe müssen dann auch alle Fakten mit dem gleichen Prädikat auftauchen. Wenn Sie es versäumen, Klauselgruppen zu bilden, werden Sie durch eine Fehlermeldung von Prolog darauf hingewiesen. Prädikatgruppen können in beliebiger Reihenfolge auftreten.

- Innerhalb einer speziellen Prädikatgruppe beginnt Prolog mit der Überprüfung für eine Unifikation mit dem ersten Faktum oder Regelkopf, das oder der mit der angegebenen Zielaussage unifiziert. Weitere Klauseln mit dem gleichen Prädikat, das auch unifiziert, werden in der Reihenfolge überprüft, in der sie im Programm vorkommen.

- Wenn ein Regelkopf mit der Zielaussage unifiziert, dann wird der Regelrumpf zu einem neuen Subziel, das dann als nächstes erreicht werden muß. Subziele werden von links nach rechts überprüft.

Die Ausführungsrichtung verläuft von oben nach unten und von links nach rechts. Prolog wendet dabei das *Tiefensuchverfahren* an, d.h. zunächst wird eine Regel so weit überprüft, bis kein Subziel mehr erreicht werden kann. Erst dann kommt die nächste Regel an die Reihe, die überprüft wird, um zu sehen, ob sie mit dem angegebenen Ziel unifiziert.

Wenn Sie die Klauseln in unserem medizinischen Diagnose-System anders anordnen, werden Sie sehen, daß das Programm immer noch funktioniert, solange Sie nur alle Prädikate der Form *symptom(Name,Indikation)* zusammenlassen und die Prädikate der Form *annahme(Name,Krankheit)* ebenso. Die Effektivität der Suche kann jedoch durch die Reihenfolge beeinflußt werden. Wenn Ihr Programm 300 Angaben für Krankheiten enthält, sollten Sie dafür sorgen, daß diejenigen, die am seltensten auftreten, am Ende des Programms stehen. Wenn z.B. die Regel mit den Angaben für eine normale Erkältung am

Ende des Programms steht, dann muß das Programm jedesmal alle anderen (vielleicht selten vorkommenden) Krankheiten überprüfen. In umfangreichen Programmen kann dieser Zeitfaktor durchaus von Bedeutung sein. Bei den meisten Anwendungen werden die Angaben über den Patienten nicht als Fakten einprogrammiert (wie in unserem Beispiel), sondern über einen Dialog mit dem Anwender erfaßt. Wenn die Klauseln dann nicht in einer sinnvollen Reihenfolge angeordnet sind, muß der Benutzer viele Fragen beantworten, die mit dem eigentlichen Problem wenig zu tun haben.

Die Definition der Prädikate und die Reihenfolge der Klauseln legen für das Programm eine Heuristik fest. Generell bezeichnet man Verfahren, die durch das Einengen des Problemraums zu einer möglichen Lösung führen, als *heuristische Verfahren*. Die meiste Zeit bei der Entwicklung eines Prolog-Programms wird darauf verwendet, die bestmögliche Heuristik für einen bestehenden Problemraum herauszufinden.

Ab Kapitel 7 werden Sie Beispiele finden, in denen Techniken zur Programmablauf-Steuerung angewendet werden und die demonstrieren, daß die Reihenfolge der Klauseln eine über die Effizienz eines Programms hinausgehende Bedeutung hat. In manchen Fällen würde eine andere Anordnung der Klauseln das Programm sogar scheitern lassen. Die Reihenfolge der Klauseln kann sogar von größerer Wichtigkeit sein als die Heuristik; sie kann die gesamte Vorgehensweise beeinflussen.

Standard-Prädikate

Die meisten Prolog-Versionen verfügen über eine Vielzahl von eingebauten Standard-Prädikaten, die der Unterstützung zahlreicher Funktionen, wie etwa zur Ablaufkontrolle, Datenein- und -ausgabe, dienen. Turbo Prolog bildet hier keine Ausnahme. Es bietet Dutzende von Standard-Prädikaten, die nicht nur Ein- und Ausgabeoperationen, sondern auch Grafik-, Datei-, Zeichenketten- und Umwandlungsoperationen unterstützen. Diese Prädikate können genau wie alle anderen Prädikate in Regeln und Fakten verwendet werden. Eine Definition im Abschnitt predicates ist für sie nicht erforderlich, da sie ein fester Bestandteil von Turbo Prolog sind. Man kann diese eingebauten Prädikate in neun Gruppen aufschlüsseln:

1. *Prädikate zur Steuerung* — steuern den Programmablauf, erzwingen oder verhindern Backtracking.
2. *Prädikate für Leseoperationen* — unterstützen das Einlesen von Daten vor der Tastatur oder aus einer Datei in eine Variable.

3. *Prädikate für Schreiboperationen* – unterstützen das Schreiben von Daten auf Bildschirm, Drucker oder in eine Datei.

4. *Prädikate für Dateisysteme* – helfen, Diskettendateien vom Turbo Prolog-Programm aus zu verwalten.

5. *Prädikate für die Bildschirmverwaltung* – unterstützen die Grafikausgabe (Fenster, Turtle-Grafik usw.) und die Tonausgabe.

6. *Prädikate für Zeichenkettenverarbeitung* – unterstützen das Arbeiten mit Zeichenketten.

7. *Prädikate für Typumwandlungen* – helfen bei der Umwandlung von Datentypen.

8. *Prädikate auf Systemebene* – ermöglichen den Zugriff auf DOS-Funktionen von einem Turbo Prolog-Programm aus.

Diese verschiedenen Gruppen werden in den folgenden Kapiteln noch näher besprochen.

Das Prädikat not

Es kann vorkommen, daß Sie in der Datenbank ausdrücklich vermerken wollen, daß ein bestimmtes Faktum nicht zutreffen soll. Zu diesem Zweck können Sie das Standard-Prädikat *not* verwenden. Das Prädikat *not* darf nicht bei der Formulierung eines Faktums oder in einem Regelkopf erscheinen. Es kann nur in einer Bedingung eingesetzt werden, wie z.B.:

```
ersetze(rechter_lautsprecher):-
not(ist(rechter_lautsprecher,funktionstüchtig)).
```

In diesem Fall wird, wenn

```
ist(rechter_lautsprecher,funktionstüchtig).
```

in der Datenbank ist, die Regel nicht zutreffen.

Sie müssen bei Ihrer Programmplanung genau zwischen der Verwendung des Prädikats *not* und dem Weglassen von Fakten in der Datenbank unterscheiden. Wenn ein Faktum nicht Bestandteil der Datenbank ist, wird es von Prolog als falsch eingestuft. Die Zuordnung des Wahrheitswertes falsch aufgrund von

Nichtvorhandensein in der Datenbank ist jedoch der Beweis für eine *not*-Bedingung. Wenn in unserem Beispiel das ist-Faktum nicht vorhanden wäre, wäre das auch noch kein Beweis für die *not*-Bedingung in der Regel.

Übungen

1. Bestimmen Sie den Kopf und den Rumpf der folgenden Regeln:

a.
```
annahme(meningitis):-
symptom(fieber),
symptom(kopfschmerzen),
symptom(hautflecken).
```

b.
```
annahme(regen):-
barometer(fallend),
wind(west).
```

2. Die folgende Datenbank sei gegeben:

```
landpr('BRD').
landpr(_).
landpr(Land):-
ort(Land).
```

Mit welcher Klausel werden die folgenden Zielaussagen jeweils zuerst unifizieren?

a. `landpr(Land)`

b. `landpr('DDR')`

c. `landpr(_)`

Kapitel 6

Einfache Ein- und Ausgaben

Ein ungeübter Anwender könnte beim Einsatz des medizinischen Diagnose-Systems aus Kapitel 5 Schwierigkeiten haben. Man könnte noch mehr Regeln und Symptome hinzufügen, aber das löst nicht das Problem. Die Schnittstelle zum Anwender muß verändert werden, so daß

- der Anwender ohne Schwierigkeiten das Ziel definieren kann. Beispielsweise könnte die Eingabe des Anwenders aus einem einzigen Wort bestehen, wie etwa *los* oder *run*, um das Programm zu starten. Von diesem Moment an läuft die Programmausführung dann automatisch ab.

- spezielle Symptome vom Anwender als Antwort auf Eingabeanforderungen während des Programmlaufs eingegeben werden können; diese werden nicht vorher einprogrammiert. Das Programm könnte beispielsweise fragen, ob Rolf Fieber hat, und dann *symptom(rolf,fieber)* in Abhängigkeit von der Antwort auf wahr oder falsch setzen.

- Ergebnisse dem Anwender in deutschen Sätzen mitgeteilt werden, wie etwa:

```
Rolf hat vermutlich Röteln.
```

- Unterstriche zwischen Wörtern weggelassen und dafür Leerstellen gesetzt werden und außerdem Namen mit Großbuchstaben beginnen. (Diese Änderungen erfordern lediglich die Verwendung von Zeichenkettenvariablen, wenn Zeichenkettenwerte ausgegeben oder eingegeben werden sollen.)

In diesem Kapitel wollen wir Ihnen zeigen, wie man diese benutzerfreundliche Bedienung in das medizinische Diagnose-Programm einbaut.

Ein Dialogbeispiel

Wir wollen uns also anschauen, wie ein Prolog-Programm einen Dialog mit dem Anwender führt, Fragen stellt, Antworten liefert und vielleicht auch dem

Anwender mitteilt, wie es zu dem Ergebnis gekommen ist. In Kapitel 1 haben wir schon einige Beispiele dazu gesehen.

Der Dialog soll beispielsweise folgendermaßen ablaufen:

Wie heißt der Patient?
Rolf
Wie alt ist der Patient?
10
Hat Rolf Fieber?
ja
Hat Rolf Husten?
nein
Hat Rolf Kopfschmerzen?
ja
Hat Rolf Schnupfen?
ja
Hat Rolf Hautausschlag?
ja
Rolf hat vermutlich Röteln. Ist der Hautausschlag rot?
ja
Ist das Fieber nur leicht?
ja
Begann der Hautausschlag im Gesicht und ist nun am Körper?
ja
Sind die Gelenke steif?
ja
Sind die Lymphknoten hinter dem Ohr geschwollen?
ja
Rolf hat Röteln.

Bei Expertensystemen wird ein solcher Dialog *Befragungs-Paradigma* genannt. Diese noch ganz allgemeine Struktur kann mit fast allen Typen von Diagnose-Systemen bearbeitet werden. Zwei Dinge sind an dieser Struktur bemerkenswert:

- Die meisten Fragen können mit *ja* oder *nein* beantwortet werden. Bei einigen werden aber auch Werte an Eingabevariablen zugewiesen, wie z.B. bei der Angabe des Alters, d.h. die Variablen werden gebunden. Darauf kommen wir später noch einmal zurück. Außerdem sind in Prolog auch Multiple-Choice-Fragen zugelassen (siehe Kapitel 12).

- Das Programm stellt über manche Symptome keine Fragen (z.B. Schüttelfrost oder Bindehautentzündung), obwohl diese in Regeln vorkommen. Durch eine Heuristik ist die Suche bereits eingeengt worden, so daß Prolog unwesentliche Fragen auslassen kann.

Um ein Prolog-Programm durch Ein- und Ausgabemöglichkeiten zu ergänzen, kann man die Standard-Prädikate einsetzen, die diese Operationen unterstützen.

Prädikate für Ausgabeoperationen

Turbo Prolog verfügt über drei Ausgabe-Prädikate: *write, writedevice* und *writef*. Diese wollen wir nun genauer betrachten. Sie können selbstverständlich auch eigene Ausgabe-Prädikate unter Zuhilfenahme der Standard-Prädikate *write* und *writef* für spezielle Ausgabevorgänge definieren, z.B. für die Ausgabe von Elementen einer Liste. Benutzerdefinierte Ausgabe-Prädikate werden in Kapitel 13 vorgestellt.

Das Prädikat write

Verändern Sie zunächst einmal das Programm aus Kapitel 5, indem Sie eine neue Klausel an den Anfang setzen, die das Standard-Prädikat *write* enthält:

```
los:-
    annahme(Patient,Krankheit),
    write(Patient,' hat vermutlich ',Krankheit,'.'),nl.
```

Vergessen Sie nicht, das Wort *los* auch im Abschnitt *predicates* aufzunehmen. Compilieren Sie nun das geänderte Programm, und starten Sie es.

Wenn Turbo Prolog nach Programmstart im Dialog-Fenster Ihre Zieleingabe erfragt, brauchen Sie nur das Wort *los* einzutippen. Dieses Wort wird mit dem Kopf der neu eingefügten Regel unifizieren, und Prolog wird damit beginnen, die Zielaussage *annahme(Patient,Krankheit)* zu lösen.

Das Ziel wird auch diesmal erreicht, indem die Variable *Patient* an den Wert *rolf* und die Variable *Krankheit* an den Wert *röteln* gebunden wird. Das Prädikat *write* bewerkstelligt dann die Ausgabe dieser Variablenwerte in einem deutschen Satz:

```
Rolf hat vermutlich Röteln.
```

Diese Ausgabe kann noch verbessert werden, indem die Objekte *name* und *krankheit* als Zeichenketten-Objekte definiert werden. Dann kann Prolog melden:

```
rolf hat vermutlich röteln.
```

Das veränderte Programm (mit noch weiteren Änderungen, die wir jetzt besprechen werden) können Sie in Abb. 6.1 am Ende dieses Kapitels betrachten. Tippen Sie diese neue Version aber jetzt nicht ab. Wir werden es nach und nach im Verlauf dieses Kapitels gemeinsam erweitern.

Machen Sie sich im Moment auch noch keine Gedanken über den Zusatz *nl* nach dem Prädikat *write*. Auf seine Bedeutung werden wir im nächsten Abschnitt zu sprechen kommen. Wir wollen uns zunächst einmal das Prädikat write genauer anschauen.

Die allgemeine Form des Prädikats write lautet

```
write(E1,E2,E3,...,En)
```

wobei E1, E2, E3, ..., En Prolog-Variablen oder Objekte vom Standard-Domaintyp sind.

Alle Variablen in der Liste müssen gebunden sein, bevor das Prädikat *write* aufgerufen werden darf. Das Prädikat *write* wird ohne Verzögerung ausgeführt und immer mit dem Wahrheitswert *True* belegt. Objekte und Variablen können in den Argumenten abwechselnd auftreten.

Auch hier findet die Ausführung von links nach rechts und von oben nach unten statt. Beispielsweise führt die Klausel

```
test:-        write("Dies ist ein Beispiel "),
    write("mit mehreren write-Anweisungen.").
```

zu der folgenden Anzeige:

```
Dies ist ein Beispiel mit mehreren write-Anweisungen.
```

Steuerung

Jetzt kommen wir noch einmal auf die Buchstaben *nl* zurück, die hinter dem Prädikat write standen. Hierbei handelt es sich um ein Standard-Prädikat für den Zeilenvorschub (newline). Diese Funktion bewirkt bei der Ausgabe das gleiche wie ein Wagenrücklauf und ein Zeilenvorschub. Die Ausgabe der folgenden Anweisungen

```
write(Patient," hat vermutlich ",Krankheit,".") ,nl,
write("Ist der Ausschlag rot (j/n)?").
```

lautet:

```
rolf hat vermutlich röteln.
Ist der Ausschlag rot (j/n)?
```

Eine andere Möglichkeit, die einen Zeilenvorschub verursacht, ist die Verwendung des Prädikats *write* zusammen mit einem *Backslash*-Befehl, in unserem Fall *n*:

```
write(Patient,' hat vermutlich ',Krankheit,'.\n'),
write('Ist der Ausschlag rot (j/n)?').
```

Turbo Prolog stellt drei Backslash-Befehle zur Verfügung:

`\n`	Wagenrücklauf/Zeilenvorschub
`\t`	Tabulatorsprung
`\Zahl`	Ausgabe des Zeichens, dessen ASCII-Wert der Zahl entspricht.

Die Backslash-Befehle müssen entweder in Anführungszeichen oder in Apostrophe gesetzt werden. Sie können auch Bestandteil einer Zeichenkette für die Ausgabe sein, wie es im vorangegangenen Beispiel der Fall war.

Wenn der Backslash selbst auf den Bildschirm oder Drucker ausgegeben werden soll, so muß er doppelt eingegeben werden:

```
\\
```

Damit stehen Ihnen nun drei gleichwertige Möglichkeiten zur Verfügung, die einen Zeilenvorschub auslösen:

```
nl
write("\n")
write('\n')
```

Die Wirkung dieser drei Befehle ist völlig identisch.

Ausgabe auf den Drucker

Alle Ausgaben auf dem Bildschirm können auf einen Drucker oder in eine Datei umgeleitet werden. Zu diesem Zweck wird das Standard-Prädikat *writedevice* eingesetzt. Die folgenden Zeilen leiten die Ausgabe des Prädikats write auf den Drucker (printer) um und dann für weitere Ausgaben wieder auf den Bildschirm (screen) zurück:

```
writedevice(printer),
write('Wir drucken auf den Drucker.'),
writedevice(screen).
```

Das Prädikat writef

Es kommt zuweilen vor, daß man Ausgaben formatieren möchte. Beispielsweise kann es erwünscht sein, daß in einer Tabelle mit Zahlen die Dezimalpunkte untereinander stehen. In Turbo Prolog gibt es für formatierte Ausgaben das Standard-Prädikat *writef*. Die allgemeine Form des Prädikats *writef* lautet:

```
writef(Format,E1,E2,E3,...,En)
```

Die Formatangaben sind von der Form

```
%-m.p
```

mit der folgenden Bedeutung:

- Der Bindestrich legt Linksbündigkeit fest; er kann auch weggelassen werden, dann ist automatisch Rechtsbündigkeit voreingestellt.

m Dieser Parameter gibt die minimale Feldbreite an; er kann auch weggelassen werden.

p Dieser Parameter legt entweder die Genauigkeit einer Gleitkommazahl oder die maximale Länge einer Zeichenkette fest.

Beachten Sie, daß die Parameter *m* und *p* durch einen Punkt voneinander getrennt sein müssen.

Der Parameter *p* kann außerdem noch einen wahlfreien Suffix haben, um den numerischen Formattyp anzugeben:

f festgelegter Dezimalpunkt (Standardeinstellung).

e Exponentialschreibweise.

g Verwendung des kürzesten Formats.

Beispielsweise führt die Anweisung

```
writef("%-10 #%5.0 DM%3.2\n",hut,23,35.1).
```

zu folgender Ausgabe:

```
hut # 23 DM 35.10
```

In diesem Beispiel beinhaltet das erste Argument von writef keinen Text, sondern eine Formatierungs-Zeichenkette, die man wie folgt interpretieren kann:

1. Die Zeile beginnt mit einem linksbündigen Feld von 10 Zeichen Länge. In diesem wird das symbolische Objekt *hut* ausgegeben.

2. Danach folgt ein Leerzeichen und die Ausgabe des Nummernzeichens #.

3. Die ganze Zahl 23 wird in einem fünfstelligen Feld ohne Dezimalstellen rechtsbündig ausgegeben.

4. Es folgt wieder ein Leerzeichen und dann die Bezeichnung DM.

5. Die Dezimalzahl 35.1 wird in einem Feld mit zwei Stellen nach dem Dezimalpunkt rechtsbündig ausgegeben.

6. Zum Schluß wird ein Zeilenvorschub ausgeführt.

Prädikate für Eingabeoperationen

In dem Handbuch zu Turbo Prolog wird mit dem Begriff *domain* der Bereich bezeichnet, zu dem die Objekte einer Relation gehören können. Auf dem Gebiet der Expertensysteme hat dieses Wort jedoch eine andere Bedeutung. Dort ist ein *domain* ein definierbarer Wissenskern über ein bestimmtes Thema. In unserem Beispielsystem umfaßt dieser Bereich die Diagnosen von acht Kinderkrankheiten. Damit ist das Alter der Patienten begrenzt und die Anzahl der Krankheiten, die diagnostiziert werden können.

Im vorigen Kapitel haben wir gesehen, daß es zwei verschiedene Arten von Informationen beim Erstellen eines Prolog-Programms über ein bestimmtes Wissensgebiet gibt. Als erstes sind die allgemeinen Informationen zu nennen, die bei jedem beliebigen Problem des Bereichs zutreffen. Ein Beispiel dafür geben die Regeln in unserem medizinischen Diagnose-System. Man spricht hierbei von einer *statischen* Datenbank, d.h. sie bleibt bei jeder Programmausführung unverändert bestehen. Als zweites können wir von einer *dynamischen* Datenbank sprechen. Diese enthält Informationen für ein bestimmtes Problem. Im Beispiel des medizinischen Diagnose-Systems besteht die dynamische Datenbank aus den Fakten, die zu Rolfs speziellen Symptomen gehören.

Unser Beispielprogramm in seiner derzeitigen Form müßte für jedes andere Problem neu geschrieben und compiliert werden (die dynamische Datenbank –

für jeden Patienten gibt es ein eigenes Programm). Das ist unpraktisch und zeitraubend. Kein Anwender möchte so lange auf Antworten warten müssen. Deshalb ist es besser, wenn nur die statische Datenbank (Regeln) fester Bestandteil des Programms ist und die dynamische Datenbank (Patient und Symptome) im Dialog mit dem Anwender bei jedem Programmlauf neu erstellt wird.

Wir wissen ja bereits, wie Fragen und Ergebnisse angezeigt werden können. Nun bleibt noch zu betrachten, wie Eingaben vom Anwender entgegengenommen werden können.

Für diesen Zweck stellt Turbo Prolog mehrere Standard-Prädikate zur Unterstützung der Eingabe bereit: *readln, readchar, readint, readreal, inkey* und *keypressed*. Sie werden sehen, daß damit die meisten Eingabeoperationen bewerkstelligt werden können. Die ersten vier Funktionen arbeiten im Prinzip gleich. Wir wollen diese vier zunächst näher erläutern.

Welches der vier *read*-Prädikate jeweils zum Einsatz kommt, hängt vom Variablentyp ab, der für die Eingabe verwendet wird:

`readln`	string oder symbol (Zeichenkette oder Symbol)
`readchar`	char (einzelnes Zeichen)
`readint`	integer (ganze Zahl)
`readreal`	real (Dezimalzahl)

Das Prädikat readln

Mit dem Prädikat *readln* werden beliebige Zeichenfolgen in eine Variable eingelesen. Sie können damit beispielsweise die Klausel *symptom(rolf,fieber)* aus dem Beispielprogramm der Abb. 5.1 folgendermaßen umdefinieren:

```
symptom(Patient,fieber)-
    write("Hat ",Patient," Fieber (ja/nein)?"),
    readln(Eingabe),
    Eingabe = "ja".
```

Wenn die Regel aufgerufen wird, erfolgt eine Ausgabe der Frage, und das Programm wartet auf die Antwort. Die Antwort muß durch das Drücken der Enter-Taste abgeschlossen werden. Die Regel wird nur mit Erfolg ausgeführt, wenn der Anwender mit

```
ja↵
```

antwortet. Zeichen, die über die Tastatur eingegeben werden, erscheinen unverzüglich auf dem Bildschirm.

Bevor Sie fortfahren, sollten Sie nun die Klauseln mit der Relation *symptom* aus Abb. 5.1 entsprechend der Vorlage aus Abb. 6.1 ändern. Fügen Sie außerdem die beiden Klauseln, die mit dem Wort *los* eingeleitet werden, hinzu. Beachten Sie, wie das Prädikat *readln* zum Einlesen des Patientennamens verwendet wird. Fügen Sie auch die insgesamt drei neuen Zeilen in den Abschnitten *domains* und *predicates* hinzu und führen auch die dort notwendig gewordenen Änderungen aus. Geben Sie dann die neue Regel ein, die am Schluß des Programms steht und ein Beispiel für die Verwendung des Prädikats *readchar* gibt, auf das wir gleich zu sprechen kommen. Compilieren Sie das neue Programm, und starten Sie es.

Das Prädikat readchar

Das Standard-Prädikat *readchar* kann in ähnlicher Weise eingesetzt werden wie readln:

```
symptom(Patient,fieber):-
      write("Hat",Patient," Fieber (ja/nein)?"),
      readchar(Eingabe),
      write(Eingabe),nl,
      Eingabe= 'j'.
```

In diesem Fall ist *Eingabe* eine Variable vom Typ *char*. Denken Sie daran, daß Argumente von diesem Typ lediglich in Apostrophe und nicht in Anführungszeichen gesetzt werden. Bei der Verwendung von *readchar* wird auf dem Bildschirm über das eingegebene Zeichen kein Protokoll geführt. Aus diesem Grund haben wir die Zeile mit der Anweisung *write* und dem Befehl zum Vorrücken in eine neue Zeile hinzugefügt, damit die Eingabe auf dem Bildschirm angezeigt wird. Bei *readchar* gibt der Anwender nur ein einzelnes Zeichen ein und braucht danach nicht die Enter-Taste zu drücken.

Bei der Verwendung des Prädikats *readchar* kann der Wunsch auftauchen, ein eigenes Eingabe-Prädikat zur Anzeige der Eingabe auf dem Bildschirm zu definieren, was in anderen Klauseln dann Programmierarbeit spart. Dort schreibt man dann nur noch:

```
symptom(Patient,fieber):-
      write('Hat ',Patient,' Fieber (j/n)?'),
      antwort(Eingabe),
      Eingabe = 'j'.
```

Wenn Sie die Klauseln so schreiben, wird Prolog versuchen, die Bedingung *antwort(Eingabe)* zu prüfen, die kein Standard-Prädikat ist. Aus diesem

Grund haben wir eine Regel am Programmende vorgesehen, die bei der Prüfung der Bedingung benötigt wird:

```
antwort(Eingabe):-
readchar(Eingabe),
write(Eingabe),nl.
```

Diese Regel liest die Antwort des Benutzers ein, gibt sie auf dem Bildschirm wieder und unifiziert sie mit *Eingabe*. Dann geht Prolog zu der Regel *symptom* zurück und prüft die letzte Bedingung, um zu sehen, ob der an *Eingabe* gebundene Wert gleich 'j' ist. In diesem Fall ist der Regelkopf bewiesen: Fieber ist eines der Symptome.

Das Prädikat readint

Das Prädikat *readint* wird verwendet, um eine ganze Zahl in eine Variable einzulesen. Betrachten wir einmal ein Beispiel:

```
alterpr(Patient):-
     write("Wie alt ist der Patient?"),
     readint(Alter),
     Alter >= 12,
     write(Patient," kann nicht mit dem"),nl,
     write("vorliegenden System
     diagnostiziert werden."),nl,
     write("Es ist nur für Kinderkrankheiten ausgelegt.").
```

In diesem Fall wird das System "Wie alt ist der Patient?" fragen und dann auf die Antwort, also die Eingabe des Alters, warten. Die danach folgende Bedingung vergleicht das Alter mit einem vorgegebenen Wert. In unserem Fall wird das Alter mit der Zahl 12 verglichen; die Überprüfung liefert das Ergebnis *falsch*, wenn das Alter weniger als 12 beträgt (diese Art von arithmetischen Operationen werden wir in Kapitel 10 ausführlich behandeln).

Wenn der Test erfolgreich abgeschlossen werden konnte, fährt Prolog mit den folgenden Bedingungen fort, so daß es zur Ausgabe der Meldung als Hinweis kommt, daß das System nur für die Diagnose von Kinderkrankheiten ausgelegt ist.

Das Prädikat readreal

Das Prädikat *readreal* wird zum Einlesen von Gleitkommazahlen in Variablen verwendet. Ein System für die Inventur könnte beispielsweise die folgenden Programmzeilen enthalten:

```
preisein(Teil,Preis):-
        write("Wie hoch ist der Preis von ",Teil,"?"),
        readreal(Preis).
```

Dieses Programm gibt die Frage nach dem Preis für ein bestimmtes Teil aus. Der Anwender gibt den Preis ein, und der Wert wird an die Variable *Preis* gebunden.

Die Prädikate inkey und keypressed

Das Prädikat *inkey* liest ein einzelnes Zeichen von der Tastatur. Es hat folgende Form

```
inkey(Zeichen)
```

Wenn keine Taste gedrückt wird, scheitert das Prädikat. Wenn jedoch eine Taste gedrückt wurde, wird die Variable *Zeichen* an den Wert dieser Taste gebunden.

Ebenso kann das Prädikat *keypressed* verwendet werden, um das Drücken einer Taste zu registrieren, nur daß hierbei kein Wert übergeben wird. Das Format des Prädikats lautet einfach:

```
keypressed
```

Das Prädikat scheitert, wenn keine Taste gedrückt wurde, und ist erfolgreich, wenn der Anwender eine Taste gedrückt hat.

Verwendung von Fenstern

Während der Programmausführung können in Turbo Prolog Fenster erzeugt und verwendet werden. Damit sind Sie in der Lage, mit geringem Programmieraufwand eine hervorragende Benutzeroberfläche zu schaffen.

Über Fenster werden wir später noch ausführlich sprechen, aber ein Standard-Prädikat wollen wir bereits an dieser Stelle erwähnen, da es in unserem Programm sinnvoll eingesetzt werden könnte.

Es handelt sich um das Prädikat zum Löschen des Dialog-Fensters: *clearwindow*. Wir wollen die erste Klausel mit dem Prädikat *los* also folgendermaßen ergänzen:

```
los:-
     clearwindow,
    write("Wie heißt der Patient?"),
     readln(Patient),
     annahme(Patient,Krankheit),
     write(Patient," hat vermutlich ",Krankheit),nl.
```

Wenn Sie bis hierhin alle in diesem Kapitel besprochenen Änderungen im medizinischen Diagnose-Programm ausgeführt haben, sollte Ihr Programm dem in Abb. 6.1 gezeigten entsprechen.

```
/* MEDIZINISCHES DIAGNOSE-SYSTEM */
/* FÜR KINDERKRANKHEITEN         */
/* APRIL 1987                    */
/* BEISPIELPROGRAMM - NICHT FÜR  */
/* PROFESSIONELLEN GEBRAUCH      */
/* mit Ein- und Ausgaben         */

domains
    krankheit,indikation = symbol
    name = string

predicates
    annahme(name,krankheit)
    symptom(name,indikation)
    antwort(char)
    los

clauses
    los:-
        write("Wie heißt der Patient?"),
        readln(Patient),
        annahme(Patient,Krankheit),
        write(Patient," hat vermutlich ",Krankheit,"."),nl.

    los:-
        write("Bedaure, ich bin nicht in der Lage,"),nl,
        write("die Krankheit zu diagnostizieren."),nl.

    symptom(Patient,fieber):-
        write("Hat ",Patient," Fieber (j/n)?"),
        antwort(Eingabe),
        Eingabe = 'j'.

    symptom(Patient,hautausschlag):-
        write("Hat ",Patient," Hautausschlag (j/n)?"),
        antwort(Eingabe),
        Eingabe = 'j'.
```

Abb. 6.1: Das überarbeitete medizinische Diagnose-Programm

```
symptom(Patient,kopfschmerzen):-
   write("Hat ",Patient," Kopfschmerzen (j/n)?"),
    antwort(Eingabe),
    Eingabe = 'j'.

symptom(Patient,laufende_nase):-
    write("Hat ",Patient," eine laufende Nase (j/n)?"),
    antwort(Eingabe),
    Eingabe = 'j'.

symptom(Patient,bindehautentzündung):-
    write("Hat ",Patient," eine Bindehautentzündung
(j/n)?"),
    antwort(Eingabe),
    Eingabe = 'j'.

symptom(Patient,husten):-
    write("Hat ",Patient," Husten (j/n)?"),
    antwort(Eingabe),
    Eingabe = 'j'.

symptom(Patient,gliederschmerzen):-
    write("Hat ",Patient," Gliederschmerzen (j/n)?"),
    antwort(Eingabe),
    Eingabe = 'j'.

symptom(Patient,schüttelfrost):-
    write("Hat ",Patient," Schüttelfrost (j/n)?"),
    antwort(Eingabe),
    Eingabe = 'j'.

symptom(Patient,halsschmerzen):-
    write("Hat ",Patient," Halsschmerzen (j/n)?"),
    antwort(Eingabe),
    Eingabe = 'j'.

symptom(Patient,niesen):-
    write("Muß ",Patient," niesen (j/n)?"),
    antwort(Eingabe),
    Eingabe = 'j'.

symptom(Patient,geschwollene_drüsen):-
    write("Hat ",Patient," geschwollene Drüsen (j/n)?"),
    antwort(Eingabe),
    Eingabe = 'j'.

annahme(Patient,masern):-
    symptom(Patient,fieber),
    symptom(Patient,husten),
    symptom(Patient,bindehautentzündung),
    symptom(Patient,laufende_nase),
    symptom(Patient,hautausschlag).
```

Abb. 6.1: Das überarbeitete medizinische Diagnose-Programm (Fortsetzung)

```
annahme(Patient,röteln):-
     symptom(Patient,fieber),
     symptom(Patient,kopfschmerzen),
     symptom(Patient,laufende_nase),
     symptom(Patient,hautausschlag).

annahme(Patient,grippe):-
     symptom(Patient,fieber),
     symptom(Patient,kopfschmerzen),
     symptom(Patient,gliederschmerzen),
     symptom(Patient,bindehautentzündung),
     symptom(Patient,schüttelfrost),
     symptom(Patient,halsschmerzen),
     symptom(Patient,husten),
     symptom(Patient,laufende_nase).

annahme(Patient,erkältung):-
     symptom(Patient,kopfschmerzen),
     symptom(Patient,niesen),
     symptom(Patient,halsschmerzen),
     symptom(Patient,schüttelfrost),
     symptom(Patient,laufende_nase).

annahme(Patient,mumps):-
     symptom(Patient,fieber),
     symptom(Patient,geschwollene_drüsen).

annahme(Patient,windpocken):-
     symptom(Patient,fieber),
     symptom(Patient,hautausschlag),
     symptom(Patient,gliederschmerzen),
     symptom(Patient,schüttelfrost).

annahme(Patient,keuchhusten):-
     symptom(Patient,husten),
     symptom(Patient,niesen),
     symptom(Patient,laufende_nase).

antwort(Eingabe):-
     readchar(Eingabe),
     write(Eingabe),nl.
```

Abb. 6.1: Das überarbeitete medizinische Diagnose-Programm

Übungen

1. Geben Sie das Programm aus Abb. 6.1 ein, und probieren Sie es aus. In dem Programm steckt ein schwerwiegendes Problem? Worin besteht es, und wodurch wird es verursacht?

Kapitel 7

Programmablauf-Steuerung: Durch Scheitern zum Ziel

Das vorliegende Kapitel ist das erste von dreien, die sich mit Techniken zur Ablaufsteuerung eines Turbo Prolog-Programmes befassen. Wir wollen hier zunächst die Funktionsweise des Prädikats *fail erläutern.* Das Standard-Prädikat *fail* erzwingt das Scheitern einer jeden Regel und damit das Backtracking zu einer anderen Klausel, um eine Unifikation zu erreichen. Jedesmal, wenn das Prädikat aufgerufen wird, scheitert das überprüfte Ziel sofort, und Backtracking setzt ein. Da das Prädikat keine Argumente hat, ist das Scheitern durch das Prädikat *fail* nicht vom Variablenwert abhängig; das Prädikat führt immer zum Scheitern.

Wir wollen uns überlegen, wozu ein solches Prädikat nützlich sein kann. Dazu sehen wir uns ein paar Beispiele an. Zunächst betrachten wir ein ganz einfaches Beispiel, an dem die Wirkungsweise des Prädikats deutlich wird:

```
los:-
    test,
    write("Hierher werden Sie nie kommen.").

test:-
    fail.
```

Wenn Sie bei der Frage nach dem Ziel *los* eingeben, wird Prolog den Kopf der ersten Regel unifizieren und versuchen, die darin enthaltenen Bedingungen zu beweisen. Die Bedingung *test* wird dabei mit dem Kopf der zweiten Regel unifizieren, dessen Bedingung das Prädikat *fail* ist, und also wird die Zielaussage scheitern. Prolog wird dann zur ersten Regel zurückgehen, und das Ziel *los* wird wieder scheitern. Die Zeile mit dem Prädikat *write* wird also nie ausgeführt. Die Ausgabe sieht folgendermaßen aus:

```
Goal: los
False
Goal:
```

Probieren Sie dasselbe Programm noch einmal aus, nachdem Sie die beiden Regelrümpfe (Bedingungen) vertauscht und den Text entsprechend verändert haben:

```
los:-
    write("Hierher werden Sie kommen."),
    test.

test:-
    fail.
```

Auch dieses Mal wird das Ziel scheitern, jedoch die Zeichenkette wird zuvor ausgegeben:

```
Goal: los
Hierher werden Sie kommen.False
Goal:
```

Verwendung des Prädikats fail

In Abb. 7.1 sehen Sie ein Beispielprogramm für das Drucken einer Namensliste. Beachten Sie, daß es keinen Abschnitt *domains* enthält. Die beiden Argumente werden im Abschnitt *predicates* als Zeichenketten deklariert. Achten Sie auch darauf, wo das Prädikat *fail* im Programm plaziert ist:

```
los:-
     ort(Stadt,Land),
     writef("%-10 %2 \n",Stadt,Land),
     fail.
```

```
/* BEISPIEL FÜR DAS PRÄDIKAT FAIL */

predicates
   ort(string,string)
   los

clauses
   los:-
        ort(Stadt,Land).
        writef("%-10 %2 \n",Stadt,Land),
        fail.

 los.

    ort("Tokio","Japan").
    ort("London","England").
    ort("Amsterdam","Holland").
```

Abb. 7.1: Beispielprogramm für das Prädikat fail

Geben Sie das Programm ein, und compilieren Sie es. Wenn Sie das Programm ausführen, sehen Sie folgende Meldungen im Dialog-Fenster:

```
Goal:       los↵
Tokio       Japan
London      England
Amsterdam   Holland
```

Es wird eine vollständige Liste ausgegeben, und das Ziel wird erreicht.

Wenn das Ziel *los* eingegeben wird, unifiziert es mit dem Kopf der ersten Regel, und hier beginnt also die Prüfung. Das Prädikat *ort(Stadt,Land)* erzwingt, daß der Wert *Tokio* an die Variable *Stadt* und der Wert *Japan* an die Variable *Land* gebunden wird. Die Position dieser Klausel wird von Prolog für den Fall markiert, daß es hierher zurückkehren muß.

Nun erfolgt die Ausgabe der Variablennamen. Aufgrund der Formatierung durch das Prädikat *writef* wird die Stadt linksbündig in einem Feld von zehn Zeichen ausgegeben. Nach der Ausgabe wird ein Zeilenvorschub ausgeführt.

Als nächstes wird das Prädikat *fail* angetroffen, das die Regel zum Scheitern bringt und Prolog zwingt zurückzukehren. Da das Prädikat *writef* immer den Wahrheitswert *True* hat, geht Prolog noch weiter zurück.

In dem Prädikat *ort(Stadt,Land)* sind die Variablen wieder frei. Prolog kehrt an die zuvor markierte Position zurück und sucht eine neue Unifikation, was auch gelingt, indem die Variable *Stadt* an den Wert *London* und die Variable *Land* an den Wert *England* gebunden wird. Prolog geht nun wieder vorwärts, um die Zielaussage zu lösen. Das Prädikat *writef* führt zur Ausgabe der neuen Variablenwerte und erzeugt einen Zeilenvorschub. Danach scheitert das Ziel wieder aufgrund des Prädikats *fail*.

Prolog geht nun zum zweitenmal zurück, findet die dritte Unifikation und gibt die Werte für diesen letzten Durchgang aus. Wieder wird Backtracking durch das Prädikat *fail* erzwungen, aber nun ist kein weiteres Prädikat der Form *ort(Stadt,Land)* mehr vorhanden. Die Klausel scheitert, und Prolog geht zu der zweiten Klausel mit dem Prädikat *los* zurück. Diese ist erfolgreich, und die Zielaussage ist gelöst.

Betrachten Sie die allgemeine Programmstruktur bei einer Verwendung des Prädikats *fail*. Die Regel, die das Prädikat *fail* enthält, wird immer scheitern. Aus diesem Grund muß eine abschließende Regel hinzugefügt werden, die immer erfolgreich ist und dadurch der Funktionsdefinition Gültigkeit verleiht.

Diese zusätzliche Klausel wird *Endebedingung* genannt. Die erste Regel (oder Regeln) übernimmt die ganze Verarbeitung und erzwingt das Backtracking dadurch, daß es ein Prädikat *fail* enthält. Das letzte Faktum oder die letzte Regel in der Definition ist immer erfolgreich und beendet das Backtracking.

Wir wollen uns anhand des Programmlistings noch einmal die Schlüsselpunkte bei einer Verwendung des Prädikats *fail* klarmachen:

- Die Reihenfolge der *los*-Klauseln ist von großer Bedeutung. Durch sie wird ein Algorithmus oder eine Prozedur festgelegt. Wenn wir die beiden Klauseln vertauschen, ist das Ziel sofort erreicht, und es wird keine Liste ausgegeben.

- Die zweite *los*-Klausel (ein einfaches Faktum) stellt einen wichtigen Teil der Definition dar. Sie beendet das Backtracking und wird deshalb *Endebedingung* genannt.

- Die Variablen der Klausel werden jedesmal freigegeben, wenn die Regel scheitert. Nach dem Backtracking werden sie aufs neue gebunden.

- Prolog *merkt* sich, welche Werte bereits gebunden waren. Es setzt bei jeder Bindung eine Markierung, so daß es bei der Rückkehr mit der nächstfolgenden Klausel fortfährt.

- Prolog geht bei der Verarbeitung von links nach rechts vor, und zwar jedesmal so weit wie möglich, bevor es zurückgeht und einen anderen Lösungsweg sucht. In unserem Beispiel werden also erst alle Möglichkeiten geprüft, um die erste *los*-Klausel zu beweisen, bevor Prolog zur zweiten Klausel übergeht.

Es kann vorkommen, daß zwei oder mehr Regeln, die alle mit dem Prädikat *fail* enden, den gleichen Kopf haben. Diese Gruppe von Prädikaten wird dann vom einem letzten Faktum oder einer letzten Regel ohne *fail*-Prädikat abgeschlossen.

Nachdem alle möglichen Lösungen der ersten Regel ausgeschöpft sind, kommt die nächste Regel mit dem gleichen Kopf an die Reihe, bis auch hier alle Lösungen gefunden wurden.

Diese Vorgehensweise wird so fortgesetzt, bis Prolog auf die letzte Klausel – ein Faktum oder eine Regel – trifft, in dem oder in der kein *fail*-Prädikat vorhanden ist. Diese Technik können Sie sich zunutze machen, um dem Programm aus Abb. 7.1 eine Überschrift voranzustellen (Abb. 7.2).

```
/* BEISPIEL FÜR DIE ZWEIFACHE    */
/* VERWENDUNG DES PRÄDIKATS FAIL */

predicates
   ort(string,string)
   los

clauses
   los:-
       writef("%-10 %5 \n","STADT","LAND"),
       fail.

   los:-
       ort(Stadt,Land)
       writef("%-10 %2 \n",Stadt,Land),
       fail.

los.
ort("Tokio","Japan").
ort("London","England").
ort("Amsterdam","Holland").
```

Abb. 7.2: Beispielprogramm für das Prädikat fail mit Überschrift

Ausschluß von Objekten

Manchmal kann es sinnvoll sein, aus einer Datenbank alle Objekte mit bestimmten Merkmalen auszuschließen. Angenommen Sie brauchen eine Liste von Städte/Länder-Kombinationen mit Ausnahme von England. Eine Möglichkeit, diese Elemente auszulassen, ist durch die Verwendung des Prädikats *fail* gegeben.

Ein Beispiel für diesen Einsatz von fail ist im Programm der Abb. 7.3 gezeigt. Dort wurde in der zweiten *los*-Regel eine neue Bedingung hinzugefügt, die zum Testen des Landes dient. Sobald der Fall eintritt, daß in *ort(Stadt,Land)* die Variable *Land* an den Wert *England* gebunden wird, wird *landpr(Land)* scheitern, und Backtracking setzt ein.

Um das weitere Überprüfen von Ländern zu ermöglichen, muß eine zweite Klausel hinzugefügt werden. Auch wenn *landpr("England")* scheitert, so setzt jedoch Backtracking zu *landpr(Land)* ein, das die zweite *landpr*-Klausel überprüft, die zum Erfolg führt. Also wird die Zeile

```
London     England
```

ausgegeben, was eigentlich nicht beabsichtigt war. Damit das Programm unseren Vorstellungen gemäß arbeitet, muß der sogenannte *Schnitt* verwendet

werden. Er wird als Ausrufungszeichen geschrieben und wird auch oft, der englischen Notation folgend, *cut* genannt. Doch darauf wollen wir in Kapitel 9 näher eingehen. Die Kombination von *fail* mit dem Schnitt stellt in Prolog-Programmen eine außerordentlich leistungsfähige Steuerungsstruktur dar, wie Sie gleich sehen werden.

```
/* BEISPIEL FÜR DEN AUSSCHLUSS VON OBJEKTEN */

predicates
   ort(string,string)
   los
   landpr(string)

clauses
   los:-
       writef("%-10 %5 \n","STADT","LAND"),
       fail.

   los:-
       ort(Stadt,Land).
       landpr(Land),
       writef("%-10 %2 \n",Stadt,Land),
       fail.

   los.

   ort("Tokio","Japan").
   ort("London","England").
   ort("Amsterdam",Holland").

   landpr( "England"):-

       fail.

   landpr(_).
```

Abb. 7.3: Beispielprogramm zum Ausschluß von Objekten mittels des Prädikats fail (funktioniert so nicht)

Übungen

1. Schreiben Sie ein kurzes Programm unter Verwendung des Prädikats *fail*, um vier Adressen für Adreßaufkleber zu drucken.

2. Schreiben Sie das Programm aus Abb. 7.3 so um, daß es richtig funktioniert. Verwenden Sie dabei anstelle des Prädikats *fail* das Prädikat *not*.

3. Warum funktioniert das Programm aus Abb. 7.1 nicht, wenn man die Klauseln vertauscht?

Kapitel 8

Programmablauf-Steuerung: Rekursion

Die Rekursion gehört zu den wichtigsten Programmiertechniken in Prolog. Sie wird häufig zusammen mit dem Prädikat *fail* (Kapitel 7) oder dem *Schnitt* (Kapitel 9) angewendet. Das vorliegende Kapitel führt Sie in das Konzept von Rekursionen ein.

Das Konzept der Rekursion

Von einer *Rekursion* spricht man, wenn bei einer Definition der zu definierende Begriff zur Definition benutzt wird. In Prolog versteht man unter einer Rekursion die Tatsache, daß sich eine Regel selbst wieder aufruft.

Ein einfaches Beispiel

Als Beispiel können Sie das Programm aus Abb. 8.1 eingeben, compilieren und ausführen. Wenn Sie bei der Frage nach dem *Ziel zähl(1)* eingeben, so erhalten Sie folgende Ausgabe:

```
 1 2 3 4 5 6 7 8 True
```

```
/* BEISPIEL FÜR EINE REKURSION */

predicates
    zähl(integer)

clauses
    zähl(9).
    zähl(N):-
        write("  ",N),
        NN = N + 1,
        zähl (NN).
```

Abb. 8.1: Ein einfaches Rekursionsbeispiel

Bei der zweiten Klausel handelt es sich um ein Beispiel für eine rekursive Definition, denn sie ruft sich selbst wieder auf:

```
zähl(N):-
     (Anweisungen)
     zähl(NN).
```

Wenn die Zielaussage *zähl(N)* gemacht wird, unifiziert sie mit der zweiten Klausel. Die Zahl wird ausgegeben, und NN wird an den Wert 1 + 1, also 2, gebunden. Nun wird eine Unifikation mit *zähl(2)* angestrebt, die wieder mit der zweiten Klausel erreicht wird und zur Anzeige dieser Zahl führt. Das Ganze wird mehrfach wiederholt, und dabei wird jedesmal die entsprechende Zahl ausgegeben. Wenn dann die Zahl 9 erreicht wird, unifiziert *zähl(9)* mit der ersten Klausel, und die Rekursion ist beendet.

Das Prädikat *zähl(N)* ruft jeweils eine Kopie von sich selbst auf, die aber völlig identisch mit der ursprünglichen Klausel ist.

Das Prädikat repeat

Eines der Prädikate, die mit Rekursionen arbeiten, ist das Prädikat *repeat*. Hierbei handelt es sich zwar nicht um ein Standard-Prädikat, aber es ist so wichtig, daß Sie es in vielen Programmen brauchen werden. Sein Format lautet:

```
repeat
repeat:-
     repeat.
```

Das Prädikat *repeat* kann eingesetzt werden, wenn man das Programm dazu bringen will, durch Backtracking weitere Lösungen zu suchen. Wenn das Prädikat *repeat* in einer Regel verwendet wird, so ist es immer erfolgreich. Wenn eine danach folgende Klausel die Regel scheitern läßt, so erfolgt immer Backtracking zum Prädikat *repeat*. Das Prädikat *repeat* ist immer erfolgreich und beendet das Backtracking. Dadurch wird Prolog gezwungen, wieder auf demselben Weg vorwärts zu suchen. Wir wollen diese Funktionsweise noch näher betrachten.

Eine Anmeldeprozedur

Wir wollen nun eine Anmelderoutine schreiben, die Sie jedem Programm vorwegschicken können. Es ist allgemein üblich, dafür auch den englischen Be-

griff *logon* zu verwenden. Bei dieser Anmeldung muß jeder Benutzer seinen Namen und ein Kennwort eingeben, das dann mit einer Tabelle verglichen wird. Wenn sich das Wertepaar in der Tabelle befindet, so ist die Anmeldung erfolgreich. Wenn sich kein entsprechendes Paar in der Tabelle befindet, dann scheitert die Anmeldung. Eine erste Version des Programms ist in Abb. 8.2 wiedergegeben. Tippen Sie es ein, und compilieren Sie es. Geben Sie nach dem Start das Ziel ein:

```
Goal: logon
```

```
domains
    name,kennwort = symbol

predicates
    eingabe(name, kennwort)
    logon
    anwender(name,kennwort)

clauses
    logon:-
        clearwindow,
        eingabe(_,_),
        write("Sie sind jetzt angemeldet."),nl.

    logon:-
        write("Sie sind leider unbefugt."),nl.

    eingabe(Name,Kennwort):-
        write("Geben Sie bitte Ihren Namen ein: "),
        readln(Name),nl,
        write("Geben Sie das Kennwort ein: "),
        readln(Kennwort),nl,
        anwender(Name,Kennwort).

    anwender(heiner,zebra).
    anwender(helena,nil).
    anwender(robert,oper).
```

Abb. 8.2: Logon-Beispiel ohne Rekursion

Das Programm wird nach einem Namen und einem Kennwort fragen. Antworten Sie wie folgt:

```
Geben Sie bitte Ihren Namen ein: helena↵

Geben Sie das Kennwort ein: nil↵
```

Diese Werte wird das Programm akzeptieren und melden:

```
Sie sind jetzt angemeldet.
```

Versuchen Sie es noch einmal. Geben Sie aber jetzt ihren eigenen Namen ein, jedoch das gleiche Kennwort (*nil*). Das erste *logon*-Prädikat wird scheitern, und Prolog versucht es mit dem zweiten. Dann wird diese Meldung ausgegeben:

```
Sie sind leider unbefugt.
```

Wir wollen nun betrachten, wie das Programm arbeitet. Nach seiner Eingabe unifiziert das Ziel mit dem Kopf der ersten Regel, die Prolog nun überprüft. Die erste Bedingung wird zum neuen Ziel, das mit dem Prädikat *eingabe(Name,Kennwort)* unifiziert. Die beiden *readln*-Prädikate binden die Variablen *Name* und *Kennwort* an die vom Anwender eingegebenen Werte.

Das Prädikat *anwender(Name,Kennwort)* beinhaltet nun gebundene Variablen und unifiziert in unserer Beispieleingabe mit *anwender(helena,nil)*. Der Regelkopf *eingabe(Name,Kennwort)* ist bewiesen, was auch dem ersten *logon*-Prädikat zum Erfolg verhilft.

Bei der zweiten Beispieleingabe, bei der wir unseren eigenen Namen eingetippt haben, scheitert der Regelkopf *eingabe(Name,Kennwort)*, da die Bedingung *anwender(Name,Kennwort)* nicht unifiziert werden kann und somit scheitert. Nachdem die erste *logon*-Regel gescheitert ist, wird die zweite aufgerufen. Diese kann ausgeführt werden und druckt also die in ihr enthaltene Meldung aus.

Ein Problem bei dem vorliegenden Programm ist, daß der Anwender nur eine einzige Chance erhält, die richtigen Werte einzugeben. Wenn er versehentlich einen falschen Namen eingibt oder sich vertippt, scheitert das Programm. Mit dem Prädikat *repeat* kann diese Schwäche behoben werden.

Verwendung des Prädikats repeat

Fügen Sie, entsprechend der Abb. 8.3, das Prädikat *repeat* in das Programm ein. Compilieren Sie es, und führen Sie es dann aus, wobei Sie auch probieren sollten, absichtlich falsche Werte einzugeben. Der Dialog stellt sich dann etwa so dar:

```
Geben Sie bitte Ihren Namen ein: alfred↵

Geben Sie das Kennwort ein: zebra↵

Sie sind leider unbefugt.
Versuchen Sie es noch einmal.
Geben Sie bitte Ihren Namen ein: heiner↵


Geben Sie das Kennwort ein: zebra↵
Sie sind jetzt angemeldet.
```

```
domains
   name,kennwort = symbol

predicates
   eingabe(name,kennwort)
   logon
   anwender(name,kennwort)
   repeat

clauses
   repeat.
   repeat:-
        repeat.

   logon:-
      clearwindow,
      eingabe(_,_),
      write("Sie sind jetzt angemeldet."),nl.

   logon:-
      repeat,
      write("Sie sind leider unbefugt."),nl.
      write("Versuchen Sie es noch einmal."),nl,
      eingabe(_,_),
      write("Sie sind jetzt angemeldet."),nl.

   eingabe(Name,Kennwort):-
      write("Geben Sie bitte Ihren Namen ein: "),
      readln(Name),nl,
      write("Geben Sie das Kennwort ein: "),
      readln(Kennwort),nl,
      anwender(Name,Kennwort).

   anwender(heiner,zebra).
   anwender(helena,nil).
   anwender(robert,oper).
```

Abb. 8.3: Beispiel bfür die Anwendung des Prädikats repeat

Vollziehen Sie einmal nach, was in diesem Beispiel geschieht. Wenn eine falsche Eingabe erfolgt, geht Prolog wie im vorigen Programm auch zur zweiten logon-Regel über und überprüft diese. Es wird die Meldung

```
"Sie sind leider unbefugt. Versuchen Sie es noch einmal."
```

angezeigt.

Das Prädikat repeat ist beim ersten Durchlauf erfolgreich, da die erste repeat-Klausel (ein Faktum) immer den Wahrheitswert True liefert. Prolog gelangt also zu der Bedingung *eingabe(Name,Kennwort)* und überprüft somit den

entsprechenden Regelkopf, der darunter steht, mit den neuen vom Anwender eingegebenen Werten. Wenn die Eingabe dieses Mal richtig war, ist die zweite *logon*-Regel erfolgreich; wenn nicht, dann setzt Backtracking ein, und Prolog kehrt zum ersten *repeat*-Prädikat zurück:

```
repeat.
```

Von da an wird Prolog versuchen, einen anderen Lösungsweg zu finden, und ruft also das zweite *repeat*-Prädikat auf, bei dem es sich um eine Regel handelt:

```
repeat:-
    repeat.
```

Die Bedingung dieser Regel unifiziert mit der ersten *repeat*-Klausel, die den Wahrheitswert *True* besitzt. Demzufolge ist die Überprüfung dieser Regel erfolgreich, und Prolog geht in der Verarbeitung nun wieder vorwärts, indem es die Eingabe eines neuen Wertepaares verlangt.

Wir haben es hier also mit einer echten Schleife zu tun. Prolog geht so lange zurück und wieder vor, bis eine Lösung für *eingabe(Name,Kennwort)* gefunden und das Backtracking abgeschlossen wird.

Obgleich die Funktionsweise des *repeat*-Prädikats komplex ist, kann man den Sachverhalt doch einfach ausdrücken: *repeat* beendet immer das Backtracking und zwingt Prolog dazu, denselben Lösungsweg erneut einzuschlagen.

Grundregeln der Rekursion

Wenn Sie mit Rekursionen arbeiten, müssen Sie immer daran denken, daß Sie im Grunde genommen eine Schleife erzeugen und deshalb überlegen sollten, wie die Rekursion beendet wird. Wenn Sie das versäumen, wird der Anwender in einer Endlosschleife hängenbleiben. Daher sollte man sich folgende Regel merken:

1. Regel: Das Programm muß eine Endebedingung für die Rekursionsschleife vorsehen.

In unserem vorigen Beispiel steuert das Prädikat *eingabe(Name,Kennwort)* den Abbruch der Rekursion, wenn ein zulässiger Name und ein ebensolches Kennwort eingegeben wird. In dem Moment wird das Prädikat *eingabe* erfolgreich und damit der Regelkopf *logon*.

Was passiert, wenn keine Endebedingung vorgesehen ist, können Sie feststellen, wenn Sie das Programm aus Abb. 8.4 eingeben.

```
predicates
   zähl(integer)

clauses
   zähl (N):-
   zähl(N).
   write("  ",N),nl,
   NN = N + 1,
   zähl (NN),
   write("Hierher werden Sie nie kommen."
```

Abb. 8.4: Rekursion ohne Abbruch

Geben Sie bei der Frage nach dem Ziel *zähl(1)* ein. Das Dialog-Fenster wird sehr schnell mit Zahlen gefüllt, bis ein Stapelüberlauf auftritt. Im Message-Fenster erscheint die Meldung:

```
1002 Stack overflow. Re-configure with Setup if necessary.

  Press the SPACE bar
```

Das Programm enthielt keine Abbruchbedingung.

Es ist wichtig zu wissen, daß sich eine Klausel bei einer Rekursion nicht selbst aufruft, sondern eine *Kopie* von sich. Deshalb lautet die zweite Regel:

2. Regel: Gebundene Variablenwerte gelten nur jeweils für eine Regel oder ein Faktum. Wenn eine Kopie der Regel oder des Faktums aufgerufen wird, so werden die Variablen in dieser Kopie wie bei jedem anderen Aufruf nur durch einen erneuten Unifikationsprozeß gebunden.

Bei jeder Kopie, die in Prolog von einer Regel in einer Ebene erzeugt wird, wird ein Markierungspunkt gesetzt. Gleichzeitig werden dabei die aktuellen Variablenwerte für den Fall zwischengespeichert, daß eine Abbruchbedingung zum Verlassen der Ebenen führt. Die Markierungen und die Variablenwerte werden auf einem Stapel abgelegt. Die Werte einer Ebene werden immer auf der obersten Speicherstelle des Stacks auf die Werte der vorigen Ebene abgelegt, die deshalb mit allen schon abgelegten Werten logisch eine Position tiefer im Stapel rutschen.

Deshalb wird dieser Stapelspeicher auch als Kellerspeicher bezeichnet (im Sinne von Einkellern, in tieferen Ebenen einlagern). Dabei kann es wie im Beispiel 8.4 passieren, daß der Speicherbereich für den Stapel überläuft und das Programm mit einer Fehlermeldung abbricht.

Das zuletzt im Kellerspeicher abgelegte Element ist dasjenige, auf das zuerst wieder zugegriffen werden kann. Der Kellerspeicher ist also eine lineare Speicherstruktur (Liste) mit einer Last-In-First-Out-Zugriffsregelung.

Zurücksetzen der Rekursionsebene

Sie können das vorige Beispielprogramm durch Hinzufügen einer neuen Klausel und durch eine kleine Änderung in der bereits vorhandenen Klausel zum Laufen bringen (Abb. 8.5).

```
predicates
   zähl(integer)

clauses
   zähl(9).
   zähl (N):-
        write("  ",N),nl,
        NN = N + 1,
        zähl (NN),
        write("Hierher werden Sie nie kommen : Ebene ",N),nl.
```

Abb. 8.5: Rekursion mit Zurücksetzen der Rekursionsebene

In Abb. 8.3 haben wir eine *logon*-Prozedur vorgestellt, in der das Prädikat repeat verwendet wurde. Diese Prozedur kann derart umgeschrieben werden, daß das *repeat*-Prädikat entbehrlich wird, wie in Abildung. 8.6 gezeigt wird. In diesem Beispiel wird die zweite *logon*-Klausel dann aufgerufen, wenn der Name und das Kennwort falsch eingegeben wurden. Das leitet die Rekursion ein, bei der die erste *logon*-Klausel erneut aufgerufen wird.

Ein weiterer Einsatzschwerpunkt für Rekursionen ist die Programmablauf-Steuerung, bei der die Schleife unter Zuhilfenahme eines Zählers so oft wie gewünscht durchlaufen wird und dann endet.

Der Einsatz einer Rekursion ist allerdings nicht immer sinnvoll. Der Programmablauf ist dabei unter Umständen schwer nachvollziehbar. Außerdem wird der Speicherplatz für das Speichern der Markierungspunkte und Variablenwerte jeder Ebene verbraucht.

Wenn Sie die Anzahl der Variablen beschränken und die Verwendung von komplexen Strukturen auf ein Minimum reduzieren, können Sie jedoch den Speicherplatzverbrauch etwas eindämmen. Falls Sie Ihr Problem mit einfachem Backtracking lösen können, sollten Sie die Verwendung des *fail*-Prädikats vorziehen, da seine Funktionsweise einfacher ist und weniger Speicherplatz benötigt.

```
domains
   name,kennwort = symbol

predicates
   eingabe(name,kennwort)
   logon
   anwender(name,kennwort)

clauses

   logon:-
      eingabe(Name,Kennwort),
      anwender(Name,Kennwort),
      write("Sie sind jetzt angemeldet."),nl.

   logon:-
      write("Sie sind leider unbefugt."),nl,
      write("Versuchen Sie es noch einmal."),nl,
      logon.

   eingabe(Name,Kennwort):-
      write("Geben Sie bitte Ihren Namen ein: "),
      readln(Name),nl,
      write("Geben Sie das Kennwort ein: "),
      readln(Kennwort),nl.

anwender(heiner,zebra).
anwender(helena,nil).
anwender(robert,oper).
```

Abb. 8.6: Logon-Programm mit Rekursion

Übung

Schreiben Sie das *logon*-Beispiel so um, daß der Anwender drei Versuche hat, um seine Eingaben zu betätigen. Nach diesen drei Versuchen sollte das Programm eine abschließende Meldung ausgeben, die besagt, daß der Anwender nicht zugriffsberechtigt ist.

Kapitel 9

Programmablauf-Steuerung: Der Schnitt

Dieses Kapitel will Ihnen die dritte Methode zur Steuerung des Programmablaufs näherbringen: den Schnitt. Der Schnitt, auch dem englischen Sprachgebrauch folgend *cut* genannt, ist eine der wichtigsten und zugleich auch komplexesten Funktionen in Prolog. Bevor Sie in Ihren Programmen mit dem Schnitt arbeiten, sollten Sie das Prinzip des Backtrackings voll und ganz verstanden haben.

Der Schnitt

Der Hauptzweck des Schnitts ist es, das Backtracking in Abhängigkeit von einer bestimmten Bedingung zu verhindern oder zu blockieren. Ein Schnitt wird als Aufrufungszeichen (!) geschrieben und hat keine Argumente. Er ist immer erfolgreich und fungiert dann wie ein Zaun, der wirkungsvoll jegliches Backtracking jenseits des Schnitts verhindert. Wenn irgendeine Bedingung, die nach dem Schnitt kommt, scheitert, kann Backtracking nur bis zu diesem Schnitt durchgeführt werden, um einen anderen Lösungsweg zu suchen. Wenn die ganze Regel scheitert, in der ein Schnitt die letzte Bedingung ist, so wird keine andere Regel mit dem gleichen Regelkopf überprüft. Prolog muß das Scheitern oder den Erfolg des Prädikats hinnehmen, das auf dieser speziellen Klausel gründet.

Beispiel für einen Schnitt

Wegen der Komplexität und der Leistungsfähigkeit des Schnitts werden wir in diesem Kapitel zahlreiche Beispiele anführen. Beim ersten handelt es sich um ein einfaches Programm ohne Rekursion. Wir greifen dazu das Beispiel aus Abb. 7.2 auf, mit dem eine Liste von Städten und Ländern ausgegeben werden konnte.

Erinnern Sie sich daran, wie mit Hilfe des Prädikats *fail* das Programm verändert wurde, so daß die Städte in England ausgelassen werden sollten (Abb. 7.3). Diese Änderung bewirkte allerdings nicht das Gewünschte. Erst wenn wir einen Schnitt setzen, funktioniert es richtig (Abb. 9.1).

```
landpr("England"):-
  !,fail.
landpr(_).
```

```
/*BEISPIEL FÜR EINEN SCHNITT */

predicates
   ort(string,string)
   los
   landpr(string)

clauses
   los:-
        writef("%-10 %5 ^\n","STADT","LAND"),
        fail.

   los:-
        ort(Stadt,Land),
        landpr(Land),
        writef("%- 10 %2 \n", Stadt,Land),
        fail.

   los.

        ort("Tokio","Japan").
        ort("London","England").
        ort("Amsterdam","Holland").

   landpr("England"):-
       !,fail.

   landpr(_).
```

Abb. 9.1: Programm mit Schnitt zur Ausgabe einer Liste

Compilieren Sie das Programm, und führen Sie es aus. Nun arbeitet es korrekt und gibt alle Städte/Länder-Kombinationen mit Ausnahme der von England aus.

In dem ursprünglichen Beispiel ohne Schnitt unifizierte das Prädikat *landpr("England")* mit der ersten *landpr("England")*-Klausel, die daraufhin scheiterte. Das war ja noch erwünscht. Das Problem bestand darin, daß danach Backtracking zu der Klausel *landpr(_)* erfolgte. Diese war immer erfolgreich, so daß alle Elemente der Liste gedruckt wurden, ob sie nun England betrafen oder nicht.

In der neuen Programmversion in Abb. 9.1 wird das alte Ablaufmuster durch den Schnitt verändert. Wenn die Regel *landpr*("England") scheitert, wird

durch den Schnitt das Backtracking zur Bedingung *landpr(Land)* verhindert und damit eine erneute Überprüfung. Durch den Schnitt wird Prolog gezwungen, sich auf die bis hier gefundenen Lösungen zu beschränken.

Wenn der Regelkopf *landpr("England")* scheitert, wird Prolog nicht weiter versuchen, *landpr* mit dem Wert *"England"* zu beweisen, und geht also zu *ort(Stadt,Land)* zurück, um mit neuen Variablenwerten fortzufahren. Die ausgegebene Liste wird demzufolge dieses Mal keine englischen Städte enthalten.

Natürlich können Sie auch andere Werte auf die gleiche Weise ausschließen. Wenn Sie beispielsweise Städte in England, Italien und Frankreich ausschließen wollen, schreiben Sie:

```
landpr("England"):-
    ,fail.

landpr("Italien"):-
    ,fail.

landpr("Frankreich"):-
    ,fail.

landpr(_).
```

Erweitern Sie die Liste der Städte und Länder im Programm, und machen Sie ein paar Probeläufe. Wenn aufgrund des Variablenwertes für das Land *landpr(Land)* mit einer der drei ersten Klauseln unifiziert wird, wird das Prädikat scheitern. Andernfalls setzt Backtracking ein, und eine Unifikation mit der letzten Klausel wird erreicht.

An diesem Beispiel werden bereits die Vorteile des Schnitts deutlich. Durch seine Verwendung wurde Prolog gezwungen, die Klausel als Entscheidung für eine Lösung mit den aktuellen Variablenwerten zu akzeptieren und keine weiteren Klauseln damit zu prüfen. Der Schnitt beendet also die Suche nach einer Lösung für das Ziel, wenn dieses in einer Regel scheitert, die den Schnitt enthält. Probieren Sie einmal aus, den Schnitt an eine andere Stelle des Programms zu setzen, z.B. nach dem Prädikat fail.

Einsatzschwerpunkte für den Schnitt

Schnitte werden hauptsächlich verwendet, um das Überprüfen bestimmter Wege auszulassen. Sie können sich die Wege in einer Datenbank als Äste eines Baums vorstellen. Wenn Sie in einem Ast keine Lösung finden konnten, werden Sie zum Stamm zurückkehren und von dort aus einen anderen Ast auspro-

bieren. Der Schnitt wirkt wie ein Zaun an der Stelle, wo er im Programm steht. Backtracking kann nicht bis hinter den Schnitt erfolgen.

In einigen Fällen (wie im letzten Beispiel) erfüllt der Schnitt eine prozedurale (algorithmische) Funktion: Das Programm funktioniert nicht ohne ihn. In anderen Fällen erfüllt er eine heuristische Funktion: Das Programm würde zwar ohne ihn funktionieren, aber durch den Schnitt werden einige Möglichkeiten ausgelassen, die von vornherein nicht zu vernünftigen Lösungen führen würden, und dadurch wird die Ausführungsgeschwindigkeit erhöht.

Betrachten Sie beispielsweise eine Regel der Form:

```
los:- Bed1 and Bed2 and ! and Bed3 and Bed4.
```

Prolog wird versuchen, *los*, wenn nötig durch Backtracking zwischen *Bedingung1* und *Bedingung2*, zu beweisen, bis beide erfüllt sind. Wenn das geschehen ist, wird der Schnitt angetroffen. Dieses Prädikat ist immer erfolgreich, also wird mit der Überprüfung von *Bedingung3* begonnen.

Wenn der Schnitt einmal überschritten ist, kann Prolog nicht mehr dahinter zurückkehren. Wenn jetzt *Bedingung3* scheitert, scheitert auch die ganze Regel. Wenn nur *Bedingung4* scheitert, setzt Backtracking bis zur *Bedingung3* ein, die dann mit anderen Werten überprüft werden kann; weiter zurückgehen kann Prolog jedoch nicht.

Wenn also entweder *Bedingung3* oder *Bedingung4* scheitert, so muß die ganze Regel als ungelöst betrachtet werden, da Backtracking bis vor den Schnitt nicht erfolgen kann. Somit können keine weiteren Lösungswege, etwa mit anderen Variablenwerten, gefunden werden.

In *Bedingung1* und *Bedingung2* sind alle Variablen gebunden, wenn das Programm auf den Schnitt trifft, und somit ist Prolog auf diese Zuordnungen festgelegt.

In unserem Beispiel wurde der Schnitt verwendet, um Backtracking zu der zweiten *landpr*-Klausel zu verhindern. Nach dem Schnitt folgte gleich das Prädikat *fail*. Da Backtracking nicht bis vor den Schnitt erfolgen konnte, scheiterte also *landpr("England")* und so auch die Bedingung *landpr(Land)*.

Also mußte das Programm bis *ort(Stadt,Land)* zurückgehen und andere Variablenwerte ausprobieren. Zusammenfassend kann man sagen, daß ein Schnitt eingesetzt werden kann, um:

- die Suche nach weiteren Lösungsmöglichkeiten für ein Ziel zu unterbinden, sobald eine bestimmte Regel (oder bestimmte Variablenwerte) überprüft wurde.
- die Suche nach weiteren Lösungsmöglichkeiten für ein Ziel zu unterbinden, nachdem das Scheitern einer bestimmten Regel (oder bestimmter Variablenwerte) mittels das Prädikats fail erzwungen wurde.
- aus Geschwindigkeitsgründen unsinnige Wege von vornherein auszulassen.

Ein Schnitt stellt jedoch nicht immer die beste Lösung dar. In vielen Fällen erfüllt das Prädikat *not* dieselbe Funktion (siehe Übung 1).

Ein weiteres Beispiel

Fügen Sie in das medizinische Diagnose-System die Änderungen wie in Abb. 9.2 ein. Lassen Sie dann das Programm einmal mit und einmal ohne den Schnitt laufen. Ohne den Schnitt wird es nicht funktionieren.

```
/* MEDIZINISCHES DIAGNOSE-SYSTEM */
/* FÜR KINDERKRANKHEITEN         */
/* APRIL 1987                    */
/* BEISPIELPROGRAMMM - NICHT FÜR */
/* PROFESSIONELLEN GEBRAUCH      */

domains
   krankheit, indikation = symbol
   name = string

predicates
   annahme(name,krankheit)
   symptom(name,indikation)
   antwort(char)
   los
   einmal
   repeat

clauses
   los:-
      repeat,
      einmal,
      write("Möchten Sie noch einen Versuch machen (j/n)?"),
      antwort(Eingabe),
      Eingabe="n".
   los.
```

Abb.9.2: Medizinisches Diagnosesystem mit Verwendung des Schnitts

```
repeat.
repeat:-
    repeat.

einmal:-
    write("Wie heißt der Patient?"),
    readln(Patient),
    annahme(Patient,Krankheit),!"
    write(Patient," hat vermutlich ",Krankheit,"."),nl.

einmal:-
    write("Bedaure, ich bin nicht in der Lage,"),nl,
    write("die Krankheit zu diagnostizieren."),nl.

symptom(Patient,fieber):-
    write("Hat ",Patient," Fieber (j/n)?"),
    antwort(Eingabe),
    Eingabe= 'j'.

symptom(Patient,hautausschlag):-
    write("Hat ",Patient," Hautausschlag (j/n)?"),
    antwort(Eingabe),
    Eingabe= 'j'.

symptom(Patient,kopfschmerzen):-
    write("Hat ",Patient," Kopfschmerzen (j/n)?"),
    antwort(Eingabe),
    Eingabe= 'j'.

symptom(Patient,schnupfen):-
    write("Hat ",Patient," Schnupfen (j/n)?"),
    antwort(Eingabe),
    Eingabe= 'j'.

symptom(Patient,bindehautentzündung):-
    write("Hat ",Patient," Bindehautentzündung (j/n)?"),
    antwort(Eingabe),
    Eingabe= 'j'.

symptom(Patient,husten):-
    write("Hat ",Patient," Husten (j/n)?"),
    antwort(Eingabe),
    Eingabe= 'j'.

symptom(Patient,gliederschmerzen):-
    write("Hat ",Patient," Gliederschmerzen  (j/n)?"),
    antwort(Eingabe),
    Eingabe= 'j'.

symptom(Patient,schüttelfrost):-
    write("Hat ",Patient," Schüttelfrost  (j/n)?"),
    antwort(Eingabe),
    Eingabe= 'j'.
```

Abb.9.2: Medizinisches Diagnosesystem mit Verwendung des Schnitts(Forts.)

```
symptom(Patient,halsschmerzen):-
     write("Hat ",Patient," Halsschmerzen (j/n)?"),
     antwort(Eingabe),
     Eingabe= 'j'.

symptom(Patient,niesen):-
     write("Muß ",Patient," niesen (j/n)?"),
     antwort(Eingabe),
     Eingabe= 'j'.

symptom(Patient,geschwollene_drüsen):-
     write("Hat ",Patient," geschwollene Drüsen (j/n)?"),
     antwort(Eingabe),
     Eingabe= 'j'.

annahme(Patient,masern):-
     symptom(Patient,fieber),
     symptom(Patient,husten),
     symptom(Patient,bindehautentzündung),
     symptom(Patient,schnupfen),
     symptom(Patient,hautausschlag).

annahme(Patient,röteln):-
     symptom(Patient,fieber),
     symptom(Patient,kopfschmerzen),
     symptom(Patient,schnupfen),
     symptom(Patient,hautausschlag).

annahme(Patient,grippe):-
     symptom(Patient,fieber),
     symptom(Patient,kopfschmerzen),
     symptom(Patient,gliederschmerzen),
     symptom(Patient,bindehautentzündung) ,
     symptom(Patient,schüttelfrost),
     symptom(Patient,halsschmerzen),
     symptom(Patient,husten),
     symptom(Patient,schnupfen).

annahme(Patient,erkältung):-
     symptom(Patient,kopfschmerzen),
     symptom(Patient,niesen),
     symptom(Patient,halsschmerzen),
     symptom(Patient,schüttelfrost),
     symptom(Patient,schnupfen).

annahme(Patient,mumps):-
     symptom(Patient,fieber),
     symptom(Patient,geschwollene_drüsen).

annahme(Patient,windpocken):-
     symptom(Patient,fieber),
     symptom(Patient,gliederschmerzen),
```

Abb.9.2: Medizinisches Diagnosesystem mit Verwendung des Schnitts(Forts.)

```
        symptom(Patient,schüttelfrost),
        symptom(Patient,hautausschlag).

    annahme(Patient,keuchhusten):-
        symptom(Patient,husten).
        symptom(Patient,niesen).
        symptom(Patient,schnupfen).

    antwort(Eingabe):-
        readchar(Eingabe),
        write(Eingabe),nl.
```

Abb.9.2: Medizinisches Diagnosesystem mit Verwendung des Schnitts (Fortsetzung)

Wenn Sie den Schnitt weglassen, tritt ein Problem auf, sobald Sie nach einer erfolgreichen Diagnose eine erneute Untersuchung einleiten wollen. Nach dem Programmstart mit der Zielaussage *los* ist die Bedingung *repeat* der ersten *los*-Klausel erfolgreich, und daraufhin wird die Bedingung einmal aufgerufen. Die Bedingung *einmal* unifiziert mit dem Kopf der ersten *einmal*-Regel.

Wir wollen nun annehmen, daß die Regel mit den vom Anwender eingegebenen Werten erfolgreich überprüft worden ist und eine Diagnose ausgegeben wird. Dann kehrt das Programm zu der *los*-Regel zurück und fährt mit der Bedingung nach *einmal* fort, die den Anwender fragt, ob er noch einen Versuch machen möchte. Wenn dieser mit *j* antwortet, scheitert die *los*-Klausel, und Backtracking setzt ein.

Beim Zurückgehen findet Prolog die erste mögliche Alternative in dem Prädikat *annahme(Patient,Krankheit)* der ersten *einmal*-Regel. Die Variablen sind nun nicht mehr gebunden, und Prolog beginnt mit der Überprüfung wieder bei der *annahme*-Regel, die es verlassen hat, als es die Lösung fand.

Mit anderen Worten, Prolog bearbeitet noch immer den ersten Patienten und muß annehmen, daß außer der Lösung, die das Programm bereits für den Patienten gefunden hat, noch eine zweite gesucht wird. Sie werden bemerken, daß das Programm auch nicht nach dem Namen des Patienten fragt.

Fügen Sie nun den Schnitt wieder in die *einmal*-Regel ein, und verfolgen Sie den Programmablauf. Sobald eine Lösung gefunden wird, sind die Variablen gebunden. Wenn Sie nun eine zweite Befragung wünschen, kann Prolog nicht nach einer anderen Krankheit forschen, da die Variable *Krankheit* immer noch gebunden bleibt. Deshalb muß das Backtracking bis zur Bedingung *re-*

peat der *los*-Regel erfolgen, bevor ein anderer Lösungsweg eingeschlagen werden kann. Die Bedingung *repeat* ist wie immer erfolgreich und ändert damit wieder Verarbeitungsrichtung von Prolog. Erneut wird die *einmal*-Klausel von Anfang an ausgeführt, so daß der Anwender nach dem Namen des zweiten Patienten gefragt wird.

Beachten Sie folgende Punkte:

- In dem vorliegenden Beispiel wird der Algorithmus durch den Schnitt beeinflußt. Wenn Sie wollen, daß das Programm für den ersten Patienten eine zweite Diagnose erstellt, müssen Sie den Schnitt weglassen. Wenn Sie das Programm jedoch für mehrere Patienten einsetzen wollen, ist der Schnitt notwendig.

- Der Schnitt erfüllt außerdem noch eine heuristische Funktion, indem durch ihn unnötige Befragungen weggelassen werden. Sobald eine Krankheit mit den entsprechenden Symptomen gefunden wird, wird die Suche beendet. Dabei spielt es eine Rolle, in welcher Reihenfolge die *annahme*-Klauseln auftreten, besonders, wenn es mehrere Krankheiten mit den gleichen Symptomen gibt.

Anmerkung: In Kapitel 11 werden wir sehen, was man dagegen tun kann, daß viele gleichlautende Fragen gestellt werden.

Gemeinsame Verwendung der Prädikate cut und repeat

In Kapitel 8 wurde das Prädikat repeat in dem logon-Beispiel zur Erzeugung einer sogenannten *repeat*-Schleife eingesetzt. Das Merkmal der *repeat*-Schleife ist, daß sie durch Scheitern abbricht. Wenn jedoch eine Klausel, die das *repeat*-Prädikat verwendet, von einer anderen Klausel aufgerufen wird, sollte die aufrufende Klausel einen Schnitt enthalten, was man wie folgt schreiben kann:

```
antwort(Eingabe):-
    repeat,
    readchar(Eingabe),
    echo(Eingabe),
   !.

echo(Eingabe):-
    Eingabe = '*',.

echo(Eingabe):-
 write(Eingabe),nl,fail.
```

```
repeat.
repeat:-
    repeat.
```

In diesem Beispiel wird das Ziel *echo(X)* scheitern, wenn nicht das *-Zeichen eingegeben wird. Nach dem Scheitern setzt Backtracking bis zum Ziel *repeat* ein, das wie immer erfolgreich ist.

Das bewirkt, daß das nächste Zeichen eingelesen und ausgegeben wird. Der Schnitt in der Definition von *antwort(Eingabe)* stellt sicher, daß die *repeat*-Schleife später nicht noch einmal ausgeführt wird.

Regel: Eine Regel, die als Bedingung ein repeat-Prädikat enthält, sollte immer einen Schnitt beinhalten, um Backtracking in die Schleife zu verhindern.

Typen von Schnitten

Wir stellen in diesem Kapitel zwei Typen von Schnitten vor. Sie heißen in Prolog *grüne* und *rote* Schnitte.

Ein Schnitt vom grünen Typ wird verwendet, um zu erzwingen, daß die Variablenwerte gebunden bleiben, sobald die richtige Klausel an der Reihe ist. Dieser Typ kam auch im medizinischen Diagnose-System der Abb. 9.2 zum Einsatz. Grüne Schnitte drücken Determinismus aus.

Ein Programm ist nicht-deterministisch, wenn durch Backtracking mehrere Lösungen erzeugt werden können.

Der rote Schnitt wird verwendet, um bestimmte Bedingungen auszulassen. Diese Art Schnitt haben wir im Beispiel der Abb. 9.1 benutzt. Dort wollten wir in einer Liste alle diejenigen Elemente weglassen, bei denen der Wert für *Land* das Wort *England* war.

Die Verwendung von Schnitten in Prolog-Programmen ist unabhängig von der Art des Typs sehr umstritten. Damit wird im Prinizip eine prozedurale Steuerung eingeführt, die in krassem Gegensatz zum deklarativen Stil eines Prolog-Programms steht.

Wenn Schnitte jedoch mit Bedacht eingesetzt werden, können sie die Übersichtlichkeit und Leistungsfähigkeit der meisten Programme verbessern. Von beiden Typen ist der grüne Schnitt vielleicht noch der annehmbarste. Ein roter Schnitt kann auch oft durch das Prädikat *not* ersetzt werden.

Determinismus

In Turbo Prolog können Sie ein Programm während des Compilierens mit der Option check_determ auf Determinismus prüfen.

Geben Sie als Beispiel das Programm aus Abb. 9.3 ein, zunächst jedoch ohne den Schnitt. Stellen Sie die Option *check_determ* an den Anfang des Programms. Compilieren Sie das Programm, und führen Sie es aus. Auf dem Bildschirm wird die folgende Meldung angezeigt:

```
4001 WARNING: non deterministic clause. (F10=Ok, Esc=abort).
```

In einigen Prolog-Versionen wird bei der Verwendung von nicht-deterministischen Klauseln Speicherplatz vergeudet. Turbo Prolog läßt allerdings nicht-deterministische Klauseln zu. Der sinnvolle Einsatz von Schnitten hilft jedoch, nicht-deterministische Klauseln zu vermeiden. Fügen Sie nun in unserem Beispiel den Schnitt an der vorgeschlagenen Stelle ein, und lassen Sie das Programm noch einmal laufen.

```
check_determ
domains
   name= string

predicates
   liesname(name)
   los

clauses
   los:-
       liesname(Name),
       write(Name),nl,
       write("Ist der Name richtig (j/n)?"),
       readchar(Eingabe),
       Eingabe = 'j',
       !.

   los:-
       nl,write("Geben Sie den Namen noch einmal
        ein."),nl,nl.
       los.

   liesname(Name):-
       write("Geben Sie den Namen des Patienten ein. "),
       readln(Name),nl.
```

Abb. 9.3: Beispielprogramm für Determinismus

Übung:

Ändern Sie das Programm aus Abb. 7.3 entsprechend den Möglichkeiten, wie sie in Abb. 9.1 gezeigt werden. Bestimmen Sie die nicht-deterministische Klausel. Wie kann dagegen vorgegangen werden?

Kapitel 10

Arithmetische Operationen

Die Verwendung von arithmetischen Operationen in Prolog unterscheidet sich wesentlich von der in prozeduralen Programmiersprachen. In Turbo Prolog kann zwar numerische Verarbeitung erfolgen, die Programmierung ist aber komplexer als in prozeduralen Sprachen. Prologs Stärke besteht jedoch darin, daß es formale Schlußfolgerungen ziehen kann, und liegt nicht in der numerischen Verarbeitung. Bei prozeduralen Sprachen ist es gerade umgekehrt; sie glänzen bei der numerischen Verarbeitung, arbeiten aber sehr uneffizient, wenn sie formale Schlußfolgerungen ziehen sollen. In diesem Kapitel wollen wir die Unterschiede zwischen den beiden Sprachtypen betrachten und zeigen, wie in Turbo Prolog die numerische Verarbeitung abläuft.

Ein einfaches Beispiel

Um den Unterschied zwischen Turbo Prolog und prozeduralen Sprachen herauszuarbeiten, wollen wir ein kleines Beispiel in Turbo Prolog eingeben:

```
predicates
    los

clauses
    los:-
        X = 4 + 3,
        write(X),nl.
```

Compilieren Sie das Programm, und starten Sie es, wobei Sie als Ziel *los* eingeben. Sie werden folgendes Ergebnis erhalten:

```
Goal: los
7
True
Goal:
```

Die erste Bedingung bindet die Variable X an die Summe von 4 und 3. Die nächste Bedingung gibt den Variablenwert aus. Bis hierhin ist noch nichts Ungewöhnliches zu bemerken. Nun versuchen Sie es einmal mit der folgenden Klausel:

```
los:-
        X = 4 + 3,
        write(X),nl,
        X = 4 - 3,
        write(X),nl.
```

Compilieren Sie das Programm, und starten Sie es. Das Ergebnis sieht so aus:

```
Goal:los
7
False
Goal:
```

Das haben wir sicher nicht beabsichtigt. Wir wollen einmal untersuchen, was passiert ist.

Das Gleichheitszeichen

Das *Gleichheitszeichen* spielt in Turbo Prolog eine andere Rolle als in anderen Sprachen (und auch als in anderen Prolog-Versionen). In dem zweiten Beispiel war die Variable X beim Aufruf der Klausel *los* frei und wurde dann in der ersten Bedingung an den Wert 7 gebunden. Dieser Wert wurde ausgegeben und ein Zeilenvorschub durchgeführt. Nachdem X nun gebunden ist, wird die nächste Bedingung normal überprüft. Es wird also getestet, ob die gebundene Variable X mit dem Wert 7 gleich dem Ergebnis der Berechnung (4 – 3), also 1, ist. Da das offensichtlich nicht der Fall ist, scheitert das Ziel.

Aus diesem Grund wird das Ergebnis der zweiten Berechnung nicht angezeigt, sondern der Wahrheitswert *False* ausgegeben, der das Scheitern des Ziels X = 4 – 3 signalisiert.

Das Gleichheitszeichen ist ein *Operator*. Die Werte links und rechts des Operators nennt man *Operanden*. In Turbo Prolog kann das Gleichheitszeichen zwei verschiedene Funktionen haben: Wenn einer der Operanden eine ungebundene Variable ist, fungiert das Gleichheitszeichen als *Zuweisungsoperator*, der die Variable bindet. Im zweiten Fall stellt das Gleichheitszeichen einen *Vergleichsoperator* dar, der einen Test durchführt.

Anmerkung: Die Funktion des Gleichheitszeichens variiert je nach Prolog-Implementation. In manchen Prolog-Versionen wird der Operator *is* als Zuweisungsoperator verwendet.

Wir können fünf Merksätze über das Gleichheitszeichen in Turbo Prolog aufstellen:

1. Wenn der eine Operand eine freie Variable und der andere ein Objekt ist, so wird die Variable an den Objektwert gebunden, und das Ziel hat Erfolg. Beispielsweise wird in dem Ausdruck X = 4 + 3 die Variable X an den Wert 7 gebunden.

2. Wenn der eine Operand eine gebundene Variable und der andere ein Objekt ist, wird das Ziel nur dann Erfolg haben, wenn der Variablenwert dem des Objekts entspricht. Ist beispielsweise die Variable X bereits an den Wert 7 gebunden und der Ausdruck X = 5 + 2 soll bewertet werden, so wird das Programm den Wert

   ```
   True
   ```

   ausgeben, da 5 + 2 gleich dem Wert der gebundenen Variablen X ist.

3. Wenn der eine Operand eine freie und der andere eine gebundene Variable ist, so erhält die freie Variable den gleichen Wert wie die gebundene Variable, und das Ziel hat Erfolg. Ist beispielsweise die Variable Y an den Wert 7 gebunden, so wird der Ausdruck X = Y die Variable X auch an den Wert 7 binden.

4. Wenn beide Operanden gebundene Variablen sind, dann wird das Ziel nur Erfolg haben, wenn beide an den gleichen Wert gebunden sind. Sind beispielsweise X und Y beide an den Wert 7 gebunden, dann wird der Ausdruck X = Y mit True bewertet, andernfalls mit False.

5. Wenn beide Operanden Objekte sind, dann wird das Ziel nur erfolgreich sein, wenn beide gleich sind. Beispielsweise wäre das bei dem Ausdruck apfel = apfel der Fall.

Anmerkung: Das Gleichheitszeichen darf nicht angewendet werden, wenn beide Operanden freie Variablen sind. Wenn beispielsweise X und Y beide ungebunden sind, dann erzeugt der Ausdruck X = Y die Meldung:

```
3009 WARNING: Two free Variables in expression.
```

Vorsicht ist auch geboten, wenn die Gleichheit von Dezimalzahlen geprüft werden soll. Dieser Test wird fehlschlagen, wenn sich die Zahlen auch nur um einen kleinen Betrag unterscheiden. Um dieses Problem zu lösen, sollten Sie zunächst herausfinden, ab welcher Dezimalstelle Ungenauigkeiten auftreten.

Das folgende Beispielprogramm wird nach Eingabe des Ziels *los* den Wahrheitswert *False* anzeigen:

```
predicates
    los

clauses
    los:-
        X = sqrt(2),
        X = 1.414,
        write(X).
```

Damit Sie die gewünschte Lösung erhalten, muß das Programm folgendermaßen geschrieben sein:

```
predicates
    los
    gleich(real,real)
clauses
    los:-
        X = sqrt(2),
        gleich(X,1.414),
        write(X).
gleich(X,Y):-
    X/Y > 0.99,
    X/Y < 1.01.
```

Beachten Sie, daß bei Dezimalzahlen, die kleiner als 1 sind, führende Nullen explizit geschrieben werden müssen.

Das Gleichheitszeichen kann auch zum Testen von zusammengesetzten Strukturen verwendet werden (siehe Kapitel 11). Zu diesem Zweck müssen Sie dann Ihr eigenes Prädikat, wie beispielsweise *gleich(X,X)*, definieren.

Vergleichsoperatoren

Vergleichsoperatoren (oder relationale Operatoren) ermöglichen es in einem Programm, bei einer Überprüfung gebundene Variablen oder Objekte zu vergleichen. In Tabelle 10.1 finden Sie eine Liste der relationalen Operatoren. Wie bereits erwähnt, kann das Gleichheitszeichen sowohl Zuweisungsoperator als auch Vergleichsoperator sein, was von den Variablenbindungen abhängig ist. Wenn Sie es als Vergleichsoperator einsetzen wollen, müssen alle Variablen gebunden sein. Geben Sie das folgende Beispiel ein:

```
predicates
    los
clauses
    los:-
    X <= 4 + 3,
  write(X),nl.
```

Dieses Programm kann noch nicht einmal compiliert werden, da X nicht gebunden ist und jeder Operand, der mit relationalen Operatoren verwendet wird, gebunden sein muß. Fügen Sie nun eine Zeile ein:

```
predicates
    los
clauses
    los:-
        X = 2,
        X <= 4 + 3,
        write(X),nl.
```

Dieses Mal kann das Programm compiliert werden. Die erste Bedingung bindet die Variable an einen Wert, der in der folgenden Zeile überprüft wird. Das Ziel schließt mit Erfolg ab.

```
Goal: los
2
True
Goal:
```

Vergleichsoperatoren können auch bei Zeichenobjekten oder -variablen, Zeichenkettenobjekten oder -variablen und Symbolobjekten oder -variablen Anwendung finden. Bei diesen Vergleichen ist ein Buchstabe *kleiner* als ein anderer, wenn er im Alphabet vor ihm steht. Beispielsweise sind die folgenden Vergleiche alle richtig:

```
"X" < "z"
"marlene" < "ulrike"
"walter" > "theo"
```

Operator	Funktion
<	kleiner als
>	größer als
=	gleich
<=	kleiner oder gleich
>=	größer oder gleich
<>	ungleich

Tabelle 10.1: Vergleichsoperatoren

Operatoren und ihre Stellung

In allgemeinen mathematischen Ausdrücken können Sie wählen, wo der Operator stehen soll, solange Sie nur bei der einmal gewählten Schreibweise bleiben. In dem Ausdruck

```
4 + 7
```

steht der Operator zwischen den beiden Operanden. Dieses Format wird *Infix-Notation* genannt. Die gleiche Relation kann auch ausgedrückt werden als

```
+(4,7)
```

Diese Schreibweise ist als *Präfix-Notation* bekannt, da der Operator den Operanden vorangeht. Ein weiteres Format ist die *Postfix-Notation*:

```
4 7 +
```

Diese Notation wird beispielsweise in Hewlett-Packard-Rechnern verwendet, außerdem bei der Sprache FORTH und ist vielen Computer-Benutzern bekannt. Die meisten Leute verwenden jedoch üblicherweise die Infix-Notation. In Turbo Prolog kann bei arithmetischen Operationen sowohl die Infix- als auch die Präfix-Notation verwendet werden.

Die meisten arithmetischen und relationalen Operationen können in Turbo Prolog direkt in der Infix-Notation ausgedrückt werden. Zusammengesetzte Ausdrücke, wie

```
X = 3 + 2
```

sind im Grunde genommen mehrere Anweisungen in Infix-Notation.

Wenn Sie jedoch ein eigenes Prädikat für eine arithmetische Operation definieren wollen, muß die Relation der Operatorname sein, und die Operanden sind die Argumente. In diesem Fall wird der Ausdruck also in Präfix-Notation geschrieben. Betrachten Sie einmal das folgende Beispiel:

```
plus(X,Y,Summe):-
     Summe = X + Y.
```

Dieses Prädikat berechnet die Summe von X und Y und stellt das Ergebnis in *Summe* zur Verfügung. Dabei wird vorausgesetzt, daß die Variablen X und Y gebunden und *Summe* frei ist.

Arithmetische Standard-Operatoren

Turbo Prolog stellt mehrere arithmetische Standard-Operatoren bereit, die alle in Infix-Notation ausgedrückt sind. Sie sind in Tabelle 10.2 aufgeführt. Diese Operatoren unterstützen die numerische Verarbeitung in Prolog-Programmen.

Dabei ist zu beachten, daß prozedurale Operatoren hier anders als in prozeduralen Programmen arbeiten; sie werden als Prolog-Funktionen verwendet, die einem Unifizierungs-Algorithmus und nicht einer festgelegten Prozedur folgen.

Verfolgen Sie beispielsweise den folgenden Dialog in Turbo Prolog. Wenn Sie ihn nachvollziehen wollen, brauchen Sie keine Klauseln oder Prädikate; Sie beginnen einfach in einem leeren Arbeitsbereich.

```
Goal: 2 + 3
416 Comparison operator expected,i.e., one of <, <=, >=,
><, <>.
Goal: 2 + 3 > 4
True
Goal: X = 2 + 3
X=5
1 Solution
Goal:
```

Operationen werden nach der Prioritätsreihenfolge der Operatoren ausgeführt. Wie in anderen Sprachen können auch hier Klammern verwendet werden, um die Reihenfolge der Berechnung von Termen eines Ausdrucks zu steuern.

Eingebaute mathematische Funktionen

Turbo Prolog enthält viele eingebaute mathematische und logische Prädikate, die bei der numerischen Verarbeitung zur Anwendung kommen können.

Eine Übersicht finden Sie in den Tabellen 10.3 und 10.4. Da es sich um Prädikate handelt, sind sie alle in Präfix-Notation geschrieben.

Symbol	Operation	Priorität	Operand1	Operand2	Ergebnis
+	Addition	4	Integer	Integer	Integer
			Integer	Real	Real
			Real	Integer	Real
			Real	Real	Real
–	Subtraktion	4	Integer	Integer	Integer
			Integer	Real	Real
			Real	Integer	Real
			Real	Real	Real
*	Multiplikation	3	Integer	Integer	Integer
			Integer	Real	Real
			Real	Integer	Real
			Real	Real	Real
/	Division	3	Integer oder Real	Integer oder Real	Real
mod	modulo-Division (Rest von Operand1/ Operand2)	2	Integer	Integer	Integer
div	Div-Division (Quotient von Operand1/ Operand1)	2	Integer	Integer	Integer
+	positives Vorzeichen	1	Integer		Integer
			Real		Real
–	negatives Vorzeichen	1	Integer		Integer
			Real		Real

Anmerkung: Die Priorität regelt die Reihenfolge der Operationen in einem Ausdruck. Dabei werden Operationen mit der Priorität 1 zuerst ausgeführt

Tabelle 10.2: Arithmetische Operatoren (Infix-Notation)

Prädikat	Funktion
abs(X)	Absolutwert von X
cos(X)	Kosinus von X (in Bogenmaß)
sin(X)	Sinus von X (in Bogenmaß)
tan(X)	Tangens von X (in Bogenmaß)
arctan(X)	Arcustangens von X
exp(X)	Exponentialfunktion
ln(X)	natürlicher Logarithmus
log(X)	Logarithmus zur Basis 10
sqrt(X)	Quadratwurzel von X
random(X)	dezimale Zufallszahl zwischen 0 und 1
round(X)	Rundungswert von X

Tabelle 10.3: Mathematische Prädikate (Präfix-Notation)

Prädikat	Funktion
bitand(X,Y,Z)	bitweise AND-Verknüpfung (Z nimmt das Ergebnis auf)
bitor(X,Y,Z)	bitweise OR-Verknüpfung
bitnot(X,Z)	bitweise Negation
bitxor(X,Y,Z)	bitweise XOR-Verknüpfung
bitleft(X,N,Z)	bitweises Schieben nach links um N Stellen
bitright(X,N,Z)	bitweises Schieben nach rechts um N Stellen

Tabelle 10.4: Logische Prädikate (Präfix-Notation)

Sie können beispielsweise den Logarithmus einer Zahl *Wert* ausgeben, wobei die Klausel wie folgt lautet:

```
ausgabelog(Wert):-
    Y = log(Wert),
    write(Y),nl.
```

Dabei gelten die gleichen Vereinbarungen wie sonst auch. Betrachten Sie beispielsweise den folgenden Ausdruck zur Ausgabe der Quadratwurzel einer Zahl X:

```
ausgabewurzel(X):-
    Y = 2,
    Y = sqrt(X),
    write(Y),nl.
```

Die Variable Y wird vor dem Aufruf der eingebauten Funktion *sqrt* gebunden. Der Ausdruck

```
Y =sqrt(X)
```

ist also ein Test. Er wird erfolgreich abgeschlossen, wenn X gleich 4 ist, mit anderen Werten wird er scheitern. Der Erfolg oder das Scheitern des Ziels hängt von dem Wert von X ab. Also handelt es sich bei der Klausel um einen Test und nicht um die Berechnung eines numerischen Wertes.

Integer-Werte können zwischen –32768 und 32767 liegen. Reelle Zahlen reichen von +/–1E-308 bis +/–1E+308.

Testen von Variablenwerten

In einigen Fällen kann es nötig sein zu prüfen, ob eine Variable bereits gebunden ist. Das Standard-Prädikat *free(X)* meldet Erfolg, wenn das Argument frei ist, und scheitert, wenn das Argument gebunden ist.

Das Prädikat *bound(X)* funktioniert gerade umgekehrt; es ist erfolgreich, wenn das Argument gebunden ist, und scheitert, wenn es noch frei ist.

Verwendung von arithmetischen Prädikaten

Achten Sie darauf, daß alle möglichen Bedingungen eines selbstdefinierten Prädikats geprüft sind, bevor Sie es für numerische Operationen einsetzen.

Erinnern Sie sich daran, daß beispielweise das Prädikat *plus*

```
plus(X,Y,Summe):-
    Summe = X + Y.
```

nur richtig arbeitet, wenn die Variablen X und Y gebunden sind und *Summe* frei ist. Das Programm kann sinnvollerweise folgendermaßen erweitert werden:

```
plus(X,Y,Summe):-
    free(Summe),
    Summe = X + Y.

plus(X,Y,Summe):-
    free(X),
    X = Summe - Y.

plus(X,Y,Summe):-
    free(Y),
    Y = Summe - X.
```

Dieses erweiterte Format ermöglicht die Berechnung jedes freien Argumentwertes, sofern nur die anderen beiden gebunden sind.

Zähler

Zähler werden in Prolog anders als in anderen Sprachen implementiert. Beispielsweise wird durch den folgenden Ausdruck kein Zähler gebildet:

```
X = X + 1
```

In Prolog darf nicht auf beiden Seiten des Gleichheitszeichens eine freie Variable stehen. Wenn andererseits X bereits gebunden ist, wird die Überprüfung scheitern.

Der zulässige Weg, einen Zähler zu implementieren, wurde bereits in Kapitel 8 erwähnt, und zwar über eine Rekursion:

```
zähl(9).

zähl(X):-
    Y = X + 1,
    zähl(Y).
```

Eine Verzögerungsschleife kann mit dem folgenden Zählertyp erzeugt werden:

```
pause(0):-.

pause(X):-
    Y = X - 1,
    pause(Y).
```

Wenn jetzt die Bedingung

```
pause(10)
```

aufgerufen wird, unifiziert sie mit der zweiten Klausel, die dann ihrerseits *pause(9)* aufruft. Dieser Wert wird so oft heruntergezählt, bis *pause(0)* erreicht ist. Diese Bedingung unifiziert dann mit der ersten Klausel, die die Rekursion beendet.

Besonderheiten

Turbo Prolog kann im Gegensatz zu den meisten anderen Prolog-Versionen keine arithmetischen Operationen ausführen, die auf Unifikationen basieren. Beispielsweise ist das Folgende nicht erlaubt:

```
los:-
    N = 20,
    leerzeichen(80 - N).

leerzeichen(0):-.

leerzeichen(X):-
    write(" "),
    NN = X - 1,
    leerzeichen(NN).
```

In Turbo Prolog muß das Programm folgendermaßen aussehen:

```
los:-
    N = 20,
    leerzeichen_rechts(N).

leerzeichen_rechts(N):-
    Y = 80 - N,
    leerzeichen(Y).

leerzeichen(0):-.

leerzeichen(X):-
    write(" "),
```

```
NN = X - 1,
  leerzeichen(NN).
```

Aus dem gleichen Grund wird auch z. B. *prüf(volt,200)* in Turbo Prolog nicht mit *prüf(volt,V>150)* unifizieren. In den meisten anderen Prolog-Versionen wird das jedoch der Fall sein.

Typumwandlungen

Turbo Prolog stellt mehrere Prädikate für Typumwandlungen zur Verfügung:

char_int(Zeichen,Integer)	bindet das Integer-Argument an den ASCII-Wert des Zeichens oder das Zeichen-Argument an den ASCII-Code der Integer-Zahl, wobei ein Argument gebunden sein muß. Wenn beide gebunden sind, prüft das Prädikat, ob der ASCII-Wert des Zeichens mit der Integer-Zahl übereinstimmt.
str_char(Zeichenkette,Zeichen)	wandelt eine Zeichenkette in ein Zeichen oder ein Zeichen in eine Zeichenkette um oder prüft beide.
str_int(Zeichenkette,Integer)	wandelt eine Zeichenkette in eine Integer-Zahl oder eine Integer-Zahl in eine Zeichenkette um oder prüft beide.
str_real(Zeichenkette,Real)	wandelt eine Zeichenkette in eine Dezimalzahl oder eine Dezimalzahl in eine Zeichenkette um oder prüft beide.

Umwandlungen zwischen Zeichenketten und Elementen vom Typ *symbol* oder zwischen ganzen Zahlen und Dezimalzahlen werden automatisch, ohne besondere Prädikate, vollzogen.

Erzeugen von Zufallszahlen

Turbo Prolog verfügt über einen Zufallszahlen-Generator in Form des Standard-Prädikats *random(Z)*. Das folgende Prädikat zeigt die Verwendung dieses Standard-Prädikats:

```
zufall(X,Y):-
   random(Z),
   Y = (X - 1) * Z + 1.
```

Das Argument X muß gebunden sein. Wenn Y frei ist, bewirkt das Prädikat, daß Y an eine Zahl zwischen 1 und X gebunden wird. Wenn Y gebunden ist, schließt eine Überprüfung in 1/X Fällen mit Erfolg ab.

Übungen

1. Definieren Sie ein Prädikat, das das Minimum der Werte X und Y liefert.

2. Geben Sie eine Definition für das Prädikat *tab(N)* an, das mittels Rekursion N Leerstellen ausgibt.

Kapitel 11

Verwendung von zusammengesetzten Objekten

In Turbo Prolog können komplexe Strukturen in Form von Listen und zusammengesetzten Objekten gebildet werden. In diesem Kapitel werden wir die Verwendung von zusammengesetzten Objekten behandeln.

Zusammengesetzte Objekte

In Turbo Prolog können Objekte wieder andere Objekte enthalten. Die daraus entstehende Struktur nennt man *zusammengesetztes Objekt*. Zusammengesetzte Objekte für Adressen könnten z. B. wie folgt definiert werden:

```
adresse(name,strasse,postleitzahl,stadt)
```

Die ganze Adresse kann in einem Prädikat wie ein einziges Objekt behandelt werden. Der erste Teil eines zusammengesetzten Objektes ist der Objektname; dieser wird auch *Funktor* genannt. Jedes Objekt, das Teil der Argumentenliste ist, heißt *Komponente*. In unserem Beispiel ist *adresse* der Funktor, und *name, straße, postleitzahl* und *stadt* sind die Komponenten. Ein zusammengesetztes Objekt kann auch als *zusammengesetzter Ausdruck* bezeichnet werden.

Ein einfaches Beispiel

In Abb. 11.1 wird ein Beispielprogramm gezeigt, in dem ein zusammengesetztes Objekt verwendet wird. Dieses einfache Programm gibt die Namen aller Ingenieure derselben Stadt aus, mit Ausnahme von *Julius Frisch*.

In diesem Beispiel lautet das zusammengesetzte Objekt *adresse(strasse,postleitzahl,stadt)*. Beachten Sie, daß dieser Ausdruck ein Objekt und kein Prädikat ist, also auch nicht im Abschnitt *predicates* definiert werden mußte. Für das Objekt muß jedoch im Abschnitt *domains* ein Typ festgelegt werden.

Ebenso muß jedes einfache Objekt innerhalb des zusammengesetzten Objektes einen Typ zugewiesen bekommen. Geben Sie das Programm aus Abb. 11.1

ein, und compilieren Sie es. Starten Sie das Programm mit der Zielaussage *los*. Sie sollten die folgende Ausgabe erhalten:

```
Helmut Obst
True
```

Diese Ausgabe bedeutet, daß Helmut Obst ein Ingenieur ist, der in derselben Stadt wie Julius Frisch wohnt. Wir wollen einmal den Programmablauf betrachten.

Nach dem Start unifiziert das Ziel *los* mit der *los*-Regel, und die erste Bedingung wird überprüft:

```
person("Julius Frisch",_,adresse(_,_,X))
```

```
/* BEISPIEL FÜR ZUSAMMENGESETZTE OBJEKTE */

domains
   ort = adresse(strasse,postleitzahl,stadt)
   strasse,postleitzahl,stadt = string

predicates
   person(string,symbol,ort)
   los

clauses
   los:-
       person ("Julius Frisch",_,adresse(_,_,X)),
       person(Name,ingenieur,adresse(_,_,X)),
       write(Name),nl.

   person("Maria Teusch",ingenieur,
   adresse("Lindenweg 11","2000","Hamburg")).

   person("Helmut Obst",ingenieur,
   adresse("Torstraße 45 a","5000","Köln")).

   person("Julius Frisch",ingenieur,
   adresse("Am Alten Markt 2","5000","Köln")).
```

Abb. 11.1: Beispielprogramm mit zusammengesetztem Objekt

Danach ist folgender Ablauf festzustellen:

1. Die erste Bedingung der *los*-Klausel versucht, ein *person*-Prädikat mit drei Argumenten zu finden, wobei das erste "Julius Frisch" lautet, und

das dritte ein zusammengesetztes Objekt mit dem Namen *adresse* ist. Dieses zusammengesetzte Objekt muß aus drei Komponenten bestehen, die mit denen in der Bedingung unifizieren.

2. Das Ziel unifiziert mit

```
person("Julius Frisch",ingenieur,
adresse("Am Alten Markt 2","5000","Köln"))
```

Die Variable X ist danach an den Wert *"Köln"* gebunden.

3. Die zweite Bedingung der *los*-Regel wird nun zum neuen Ziel:

```
person(Name,ingenieur,adresse(_,_,'Köln'))
```

Nun versucht Prolog ein *person*-Prädikat mit drei Argumenten zu finden, wobei das zweite den Wert *ingenieur* haben muß. Das dritte Argument muß wieder ein zusammengesetztes Objekt mit dem Namen *adresse* sein, das aus drei Komponenten besteht. Die dritte Komponente muß *Köln* heißen.

4. Die gerade beschriebene Bedingung unifiziert mit dem zweiten *person*-Prädikat im Programm, und dabei wird die Variable *Name* an den Wert *Helmut Obst* gebunden.

5. Die letzte Bedingung der *los*-Klausel – ein write-Prädikat – wird aufgerufen und der Wert von *Name*, *Helmut Obst*, im Dialog-Fenster ausgegeben.

Dieses einfache Beispiel veranschaulicht die prinzipielle Funktionsweise bei der Verwendung von zusammengesetzten Objekten.

Gebrauch von zusammengesetzten Objekten

Die Komponenten eines zusammengesetzten Objekts müssen nicht notwendigerweise vom gleichen Typ sein. Elemente vom Typ symbol, Zeichen, Zeichenketten, ganze Zahlen, Dezimalzahlen und Listen (siehe Kapitel 13) können zusammen verwendet werden. Die Komponenten können sogar selbst wieder zusammengesetzte Objekte sein. Dadurch ist es möglich, hierarchische Objekte mit mehreren Ebenen zu definieren, z. B.:

```
fahrrad(rahmen,vorderrad(speichen,
schlauch,nabe(lager..)..)..)
```

Wir wollen nun ein paar Beispiele geben, die zeigen, wie zusammengesetzte Objekte verwendet werden können.

Anzahl der Regeln in einem Programm einschränken

In dem medizinischen Diagnose-System, das in den vorigen Kapiteln entwikkelt wurde, reicht eine einzige Klausel aus, um den Ausschlagtyp näher zu identifizieren:

```
symptompr(Symptom(Typ)):-
    write("Ist der ",Symptom),nl,
    write(Typ," (j/n)?"),
    antwort(Eingabe),
    Eingabe = 'j'.
```

Wenn die Variable *Symptom* an den Wert *Hautausschlag* und die Variable *Typ* an den Wert *fleckig mit Pusteln und Pickeln zur gleichen Zeit* gebunden ist, sieht die Anzeige so aus:

```
Ist der Hautausschlag
fleckig mit Pusteln und Pickeln zur gleichen Zeit (j/n)?
```

Sie sehen, daß diese Methode die Anzahl der Regeln im Programm reduziert (und Speicherplatz spart), da mit einer einzigen Regel viele Variationen eines Symptoms geprüft werden können.

Verbinden von Informationen zu einem Objekt

In dem Beispiel zu Beginn dieses Kapitels wurde *adresse(strasse,postleitzahl,stadt)* als zusammengesetztes Objekt definiert. Auf diese Weise kann die Adresse im Programm wie ein einziges Objekt behandelt werden, was an einen Satz in einer herkömmlichen Datenbank-Anwendung erinnert, der einen View mit Feldern aus verschiedenen Tabellen eines relationalen Datenbanksystems darstellt.

Es besteht eine enge Verwandtschaft zwischen Prolog und dem relationalen Datenbankmodell sowie der diesem zugrundeliegenden relationalen Algebra. Ein zusammengesetztes Objekt ermöglicht ebenso die Darstellung einer hierarchischen Struktur mit mehreren Ebenen, wie dies ein leistungsfähiges Datenbanksystem tut.

Sämtliche Operatoren und Funktionen der relationalen Algebra und Elemente von relationalen Datenbank-Abfragesprachen lassen sich in Prolog realisieren. Prolog und relationale Datenbank-Abfragesprachen sind dadurch gekennzeichnet, daß sie eine Lösungsmengen-orientierte (und nicht wie klassische prozedurale Sprachen nur Einzelsatz-orientierte) Arbeitsweise von ihrer Konzeption her unterstützen.

Unterscheiden zwischen mehreren Objektarten

In unserem medizinischen Diagnose-System können wir die Tatsache, daß Rolf und Hans Fieber haben, ausdrücken als

```
symptom(rolf,fieber).
symptom(hans,fieber).
```

Damit wissen wir aber noch nicht, ob sie leichtes oder mittleres Fieber haben. Das könnte dann spezifiziert werden als

```
symptom(rolf,leicht).
symptom(hans,mittel).
```

Das wiederum sagt nicht aus, ob Rolf und Hans leichtes oder mittleres Fieber oder leichten oder mittleren Husten haben. Diese Unklarheit kann durch die Verwendung eines zusammengesetzten Objekts beseitigt werden:

```
symptom(rolf,fieber(leicht)).
symptom(hans,fieber(mittel)).
```

Deklaration von zusammengesetzten Objekten

Wir wissen bereits, daß jede Komponente eines zusammengesetzten Objekts im Abschnitt *domains* einen Typ zugewiesen bekommen muß. Beispielsweise kann das zusammengesetzte Objekt unseres letzten Beispiels, wie in Abb. 11.2 gezeigt, deklariert werden. Hier wurde eine Mehrfachdefinition für *sym* verwendet. Dabei ist das Semikolon als eine Oder-Verknüpfung zu verstehen. Die verschiedenen Funktoren dürfen unterschiedlich viele Argumente besitzen.

```
domains
   sym = fieber(beschreibung);husten(typ);kopfschmerzen
   beschreibung,kopfschmerzen,typ = symbol
   patient=symbol

predicates
   symptom(patient,sym)

clauses
   symptom(rolf,fieber(leicht)).
   symptom(hans,fieber(mittel)).
   symptom(susi,husten(kratzend)).
   symptom(anne,kopfschmerzen).
```

Abb. 11.2: Zusammengesetztes Objekt mit Komponenten verschiedenen Typs

Man kann eine Hierarchie von Objekten bilden, bei der ein Objekt, das eine Komponente eines zusammengesetzten Objekts ist, selbst wieder als zusammengesetztes Objekt definiert wird, z. B.:

```
symptom(herbert,hautausschlag(farbe(rot)))
```

Auch dieses Mal muß wieder jedes zusammengesetzte Objekt im Abschnitt *domains* mit einem Typ versehen werden, auch wenn es Teil eines anderen zusammengesetzten Objekts ist. Die Änderung im Abschnitt *domains* lautet:

```
domains
    sym = fieber(beschreibung);
    hautausschlag(farbe);kopfschmerzen
    farbe = farbe(farb)
    beschreibung,kopfschmerzen,farb = symbol
    patient = symbol
```

Eingabe von Daten in ein zusammengesetztes Objekt

Beim Einlesen von Daten in Variablen von zusammengesetzten Objekten kommen Sie mit einem einzigen Eingabe-Prädikat nicht aus. Zu jeder Variablen des zusammengesetzten Objekts gehört ein eigenes Eingabe-Prädikat. Abb. 11.3 zeigt in einem Beispiel, wie man zusammengesetzte Objekte einliest.

```
domains
  person = adresse(name,strasse,postleitzahl,stadt)
  name,strasse,postleitzahl,stadt = string)
predicates
  liesadresse(person)
  los
clauses
  los:-
      liesadresse(Adresse),nl,
      write(Adresse),nl,nl,
      write("Richtig (j/n)? "),
      readchar(Eingabe),
      Eingabe = 'j',!.
  los:-
      nl,write("Bitte neue Eingabe"),nl,
      los.
  liesadresse(adresse,(Name,Strasse,Postleitzahl,Stadt)):-
      write("Name:     "),readln(Name),
      write("Straße:   "),readln(Strasse),
      write("PLZ:      "),readln(Postleitzahl),
      write("Stadt:    "),readln(Stadt).
```

Abb. 11.3: Eingabe von Daten in ein zusammengesetztes Objekt

Übungen

1. Benennen Sie die Funktoren und die Komponenten in den folgenden Ausdrücken:

a. `auto(baujahr,fabrikat,farbe)`

b. `teil(nummer,beschreibung,bestand,preis)`

c. `ort(stadt,land)`

d. `computer`

2. Schreiben Sie ein kurzes Programm zur Eingabe von Lagerbeständen unter Verwendung des zusammengesetzten Objekts *teil(nummer,beschreibung,bestand,preis)*.

Kapitel 12

Verwendung von dynamischen Datenbanken

Mit dem medizinischen Diagnose-System in Kapitel 6 gab es noch einige Schwierigkeiten. Beim Dialog mit dem System traten immer wieder doppelte Fragen auf, z. B.:

```
Hat Rolf Fieber (j/n)?
n
Hat Rolf Fieber (j/n)?
n
Hat Rolf Fieber (j/n)?
n
Hat Rolf Kopfschmerzen (j/n)?
j
```

Das Programm verfügte nicht über die Möglichkeit, sich Antworten aus vorherigen Fragen zu merken. Dadurch arbeitete es sehr uneffizient. Wir wollen nun sehen, wie man dieses Problem beseitigen kann.

Das medizinische Diagnose-Programm

Zunächst betrachten wir noch einmal das Programm aus Abb. 6.1 und seinen Ablauf. Wir wollen annehmen, daß der Patient eine normale Erkältung hat. Nach dem Programmstart unifiziert das Ziel *los* mit der ersten *los*-Klausel und möglicherweise mit dem Regelkopf *annahme(Patient,masern).*

Die Überprüfung dieser Regel beginnt mit der Frage, ob der Patient Fieber hat. Wenn wir die Frage verneinen, setzt Backtracking ein, und die nächste Regel mit dem Kopf *annahme(Patient,röteln)* wird überprüft. Wieder überprüft Prolog die erste Bedingung, *symptom(Patient,fieber)*, und fragt, ob der Patient Fieber hat.

Die Frage wird auch dieses Mal verneint, die Regel scheitert, und nun kommt die Regel *annahme(Patient,grippe)* an die Reihe. Dieser Ablauf wird so lange fortgesetzt, bis Prolog die Regel *annahme(Patient,erkältung)* erreicht.

Dieses Mal lautet die erste Bedingung *symptom(Patient,kopfschmerzen)*, und folglich wird als erstes gefragt, ob der Patient Kopfschmerzen hat. Bei der Bearbeitung der verschiedenen Regeln hat Prolog mehrfach die gleichen Fragen gestellt, wenn sie für die jeweilige Regel wichtig waren, da sich das Programm die gegebene Antwort nicht von einer Regel bis zur nächsten merken konnte. Was dem Programm fehlt, ist eine Art Datenbank, in der Fakten über ein bestimmtes Problem während der Befragung gespeichert werden können.

Zwei Typen von Datenbanken

Ein Prolog-Programm ist, wie wir bereits wissen, eine Sammlung von Fakten und Regeln über ein bestimmtes Wissensgebiet. Das Programm ist im Grunde genommen eine Datenbank, und Prolog ist eine sehr leistungsfähige Abfragesprache, mit der Fakten aus dieser Datenbank über einen Unifikations-Algorithmus ausgewählt werden können.

Das Programm ist jedoch eine *statische* Datenbank, d. h. diese Datenbank verändert sich nicht. Wenn das Programm compiliert wird, werden die Informationen in der Datenbank regelrecht "eingefroren".

Eine bessere Lösung für das medizinische Diagnose-System könnte durch Hinzufügen einer *dynamischen* Datenbank zum Programm gefunden werden. Mit Hilfe von Standard-Prädikaten können dann während des Programmablaufs Fakten zu dieser dynamischen Datenbank hinzugefügt oder aus ihr entfernt werden.

Die statische Datenbank

Eine statische Datenbank in Prolog unterscheidet sich deutlich von einer Datenbank, wie man sie bei herkömmlicher Programmierung verwendet. In anderen Programmiersprachen (wie BASIC oder dBASE III) werden zwar auch Informationen über bestimmte Dinge in der Datenbank gespeichert, aber alle Regeln, die die Beziehungen zwischen den einzelnen Elementen herstellen, sind Teil des prozeduralen Programms.

Wir wollen einmal das Beispiel betrachten, daß ein Hersteller für seine Firma , die mit der Herstellung von Fahrrädern befaßt ist, ein Lagerbestands-Kontrollsystem braucht. Jedes Fahrradmodell stellt eine hierarchische Struktur von Zubehör, Unterzubehör und Teilen dar (Abb. 12.1). Zum Inventar der Firma gehören daher Modelle, Zubehör, Unterzubehör und Teile. In dBASE III könnte der Hersteller eine Datenbank-Datei erzeugen, in der jedes Modell, das Zubehör, das Unterzubehör und die Teile zusammen mit dazugehörenden

wichtigen Informationen (Bestand, Preis, Beschreibung usw.) gespeichert werden. Er könnte sogar eine separate Datenbank-Datei mit Informationen darüber, welche Teile in welchem Unterzubehör und welche Unterzubehörteile zu bestimmten Zubehörteilen gehören, anlegen.

In der Realität ist das Problem des Herstellers noch wesentlich umfangreicher. Ein bestimmtes Fahrradmodell kann es in verschiedenen Größen und Ausführungen geben. Die Liste der Zubehörteile, Unterzubehörteile und Teile würde dann noch von der Größe des Fahrrads und anderen Faktoren abhängen. Mit anderen Worten, zahlreiche IF-THEN-Regeln kontrollieren die Teileliste für jedes einzelne Modell. Dieser Programmteil ist genauso dynamisch wie die Teileliste und kann nicht in einer Datenbank gespeichert werden. Wenn der Hersteller seine Bestandsdaten mit dBASE III erfassen möchte, müssen diese Regeln ein fester Bestandteil des Programms sein, und ein Programmierteam wäre permanent damit beschäftigt, das Programm auf dem laufenden zu halten.

Wenn es sich bei unserem Hersteller um einen Autofabrikanten handeln würde, wäre das Problem noch akuter. Die meisten Systeme der Spitzentechnologie (wie Autos, Computer und Flugzeuge) gibt es in verschiedenen Ausführungen. Diese Systeme verfügen über ein modulares dynamisches Konzept. Die Firmen, die solche Produkte herstellen, gehen intensiv auf Kundenwünsche ein und ändern ihr Produkt, wenn nötig, entsprechend den speziellen Wünschen. Die Teile sind oftmals an Hunderte oder Tausende von IF-THEN-Regeln gebunden. Wenn ein Auto eine Anhängerkupplung bekommt, kann es sein, daß die Vorderradaufhängung verbessert oder die Leistungsfähigkeit des Motors erhöht werden muß. Diese Art von Information kann nicht in einer herkömmlichen Datenbank gespeichert werden.

In Prolog können Sie jedoch beides in der Datenbank speichern, sowohl die einzelnen Informationen als auch die Regeln über die Beziehungen zwischen den einzelnen Komponenten. Hierbei handelt es sich um eine statische Datenbank, da sie sich während der Programmausführung nicht verändert. Ein Prolog-Programm besteht aus einer solchen statischen Datenbank und ein paar zusätzlichen Regeln, die den Programmablauf steuern. Um eine statische Datenbank zu verändern, muß das Programm editiert und neu compiliert werden.

In Turbo Prolog muß die gesamte statische Datenbank zur gleichen Zeit in den Rechnerspeicher passen. Die maximale Größe einer Datenbank ist also durch den verfügbaren Speicherplatz beschränkt. Einige Prolog-Versionen (wie z.B. Arity Prolog) bieten dazu noch die Möglichkeit eines virtuellen Speichers, wodurch die Datenbank während der Ausführung teilweise im Speicher und teilweise auf der Platte sein kann. Der Informationstransfer von der Platte zum

Speicher bleibt dabei für den Anwender unsichtbar und wird automatisch vollzogen. Bei diesen Prolog-Versionen wird die Größe der Datenbank allein durch die Plattenkapazität beschränkt. Es kann jedoch vorkommen, daß bei diesen Prolog-Versionen, die mit virtuellem Speicher arbeiten, die Verarbeitungsgeschwindigkeit aufgrund der Zugriffszeiten auf die Platte relativ gering ist.

Da die aktuelle Version von Turbo Prolog nicht mit virtuellem Speicher arbeitet, ist sie wesentlich schneller als andere. Alle Daten befinden sich im Speicher, und daher besteht während der Programmausführung keine Notwendigkeit für Plattenzugriffe. Aus diesem Grund benötigt man für die Entwicklung von Turbo Prolog-Programmen so viel Speicherplatz wie möglich.

Die dynamische Datenbank

Während der Programmausführung können Fakten, die der Anwender im Dialog eingibt, in einer dynamischen Datenbank gespeichert werden. Das geschieht mit Hilfe von einem oder mehreren Datenbank-Prädikaten, die als solche im Abschnitt *database* des Programms definiert werden müssen. Diese Datenbank-Prädikate werden dann mit einem Standard-Datenbank-Prädikat von Turbo Prolog aktualisiert. Wir werden später sehen, wie das genau vonstatten geht.

In Turbo Prolog wird die dynamische Datenbank zusammen mit der statischen Datenbank im Rechner gespeichert. Ihr Computersystem sollte also über ausreichend viel Speicher verfügen, um beide Datenbanken aufnehmen zu können. Sie können die dynamische Datenbank zwar auf Platte speichern (mit dem Prädikat *save*) und auch wieder einlesen (mit dem Prädikat *consult*), während der Programmausführung befindet sie sich jedoch zusammen mit dem Programm im Speicher.

Anmerkung: Turbo Prolog bietet auch Standard-Prädikate für den Zugriff auf wahlfreie Dateien. Sie können Fakten auf der Platte speichern und darauf mit Hilfe dieser Prädikate wieder zugreifen. (Wie man eine wahlfreie oder Direktzugriffs-Datei erstellt und verwendet, wird im Handbuch zu Turbo Prolog ausführlich erläutert.) Auf diese Art und Weise können Sie einen Teil der Datenbank als Datei ablegen und über diese Prädikate wieder ansprechen, was aber einige Programmiererfahrung erfordert. Zu beachten ist auch, daß häufige Platten-/Diskettenzugriffe die Ausführungszeit eines Programms sehr stark verlangsamen können. Die Geschwindigkeit für Zugriffe auf wahlfreie Dateien kann durch Verwendung einer RAM-Disk erhöht werden.

Typzuweisung für Prädikate dynamischer Datenbanken

Um eine dynamische Datenbank zum Speichern von Informationen verwenden zu können, müssen Sie mindestens ein Datenbank-Prädikat erzeugen. Während des Programmlaufs können dann Fakten mit Hilfe der Standard-Prädikate *asserta(Faktum)* und *assertz(Faktum)* in diesen Prädikaten gespeichert werden, oder es können Fakten aus einer Plattendatei mit dem Prädikat *consult(Dateiname)* gelesen werden.

Um Datenbank-Prädikate zu erzeugen, müssen Sie Ihr Programm um den Abschnitt *database* erweitern und dort die Prädikate definieren. Dieser Abschnitt muß gleich nach dem Abschnitt *domains* folgen, also vor dem Abschnitt *predicates* stehen. In Abb. 12.2 ist das medizinische Diagnose-System um eine dynamische Datenbank bereichert worden.

```
/* MEDIZINISCHES DIAGNOSE-SYSTEM */
/* FÜR KINDERKRANKHEITEN         */
/* APRIL 1987                    */
/* BEISPIELPROGRAMMM - NICHT FÜR */
/* PROFESSIONELLEN GEBRAUCH      */
/* mit dynamischer Datenbank     */

domains
   krankheit,symptom = symbol
   frage = string
   eingabe = char

database
   xpositiv(symptom)
   xnegativ(symptom)

predicates
   annahme(name,krankheit)
   symptom(name,indikation)
   antwort(eingabe)
   los
   positiv(frage,symptom)
   lösch_fakten
   erinnere(symptom,eingabe)
   frag(frage,symptom,eingabe)

clauses
   los:-
         clearwindow,
         annahme(Krankheit),!,
         write("Der Patient hat vermutlich
         ",Krankheit,"."),nl,
         lösch_fakten.
```

Abb. 12.2: Medizinisches Diagnose-System mit dynamischer Datenbank

```
los:-
     write("Bedaure, ich bin nicht in der Lage,"),nl,
     write("die Krankheit zu diagnostizieren."),nl.
     lösch_fakten.

positiv(_,Symptom):-
     xpositiv(Symptom),!.

positiv(Frage,Symptom):-
     not(xnegativ(Symptom)),
     frag(Frage,Symptom,Eingabe),
     Eingabe = 'j'.

frag(Frage,Symptom,Eingabe):-
     write(Frage),nl,
     readchar(Eingabe),
     write(Eingabe),nl,
     erinnere(Symptom,Eingabe).

erinnere(Symptom,'j'):-
     asserta(xpositiv(Symptom)).

erinnere(Symptom,'n'):-
     asserta(xnegativ(Symptom)).

lösch_fakten:-
     retract(xpositv( _)),fail.

lösch_fakten:-
     retract(xnegativ( _)),fail.

lösch_fakten.

symptom(fieber):-
     positiv("Hat der Patient Fieber (j/n)?",fieber).

symptom(hautausschlag):-
     positiv("Hat der Patient Hautausschlag
      (j/n)?",hautschlag).

symptom(kopfschmerzen):-
     positiv("Hat Patient Kopfschmerzen (j/n)?",
     kopfschmerzen).

symptom(schnupfen):-
     positiv("Hat der Patient Schnupfen (j/n)?" schnupfen).

symptom(bindehautentzündung):-
     positiv("Hat der Patient Bindehautentzündung (j/n)?",
     bindehautentzündung).

symptom(husten):-
     write("Hat der Patient Husten (j/n)?", husten).
```

Abb. 12.2: Medizinisches Diagnose-System mit dynamischer Datenbank (Fortsetzung)

```
    symptom(gliederschmerzen):-
        positiv("Hat der Patient Gliederschmerzen (j/n)?",
        gliederschmerzen).

    symptom(schüttelfrost):-
        positv("Hat der Patient Schüttelfrost (j/n)?",
        schüttelfrost).

    symptom(halsschmerzen):-
        positiv("Hat der Patient Halsschmerzen (j/n)?",
        halsschmerzen).

    symptom(niesen):-
        positiv("Hat der Patient Niesen (j/n)?", niesen).

    symptom(geschwollene_drüsen):-
        positiv("Hat der Patient geschwollene Drüsen (j/n)?",
        geschwollene_drüsen).

    annahme(masern):-
        symptom(fieber),
        symptom(husten),
        symptom(bindehautentzündung),
        symptom(schnupfen),
        symptom(hautausschlag).

    annahme(röteln):-
        symptom(fieber),
        symptom(kopfschmerzen),
        symptom(schnupfen),
        symptom(hautausschlag).

    annahme(grippe):-
        symptom(fieber),
        symptom(kopfschmerzen),
        symptom(gliederschmerzen),
        symptom(bindehautentzündung),
        symptom(schüttelfrost),
        symptom(halsschmerzen),
        symptom(husten),
        symptom(schnupfen).

    annahme(erkältung):-
        symptom(kopfschmerzen),
        symptom(niesen),
        symptom(halsschmerzen),
        symptom(schüttelfrost),
        symptom(schnupfen).

    annahme(mumps)
        symptom(fieber),
        symptom(geschwollene_drüsen).

annahme(windpocken):-
        symptom(fieber)
```

Abb. 12.2: Medizinisches Diagnose-System mit dynamischer Datenbank(Fortsetzung)

```
annahme(windpocken),
    symptom(fieber),
    symptom(gliederschmerzen),
    symptom(schüttelfrost),
    symptom(hautausschlag).

annahme(keuchhusten):-
    symptom(husten),
    symptom(niesen),
    symptom(schnupfen).

antwort(Eingabe):-
    readchar(Eingabe),
    write(Eingabe),nl.
```

Abb. 12.2: Medizinisches Diagnose-System mit dynamischer Datenbank

Die Anzahl der zu definierenden Datenbank-Prädikate hängt von der Anwendung ab. Ein Inventar-Programm oder eine Adressenverwaltung benötigen nur ein einziges Datenbank-Prädikat zum Speichern des Inventars oder der Adressen. Damit wäre dynamisches Hinzufügen, Ändern und Löschen von Datenbank-Elementen ohne erneute Compilierung des Programms nach jeder Änderung bereits gewährleistet. In einem Prolog-Programm zur Adressenverwaltung könnte das Datenbank-Prädikat

```
adresse(name,strasse,postleitzahl,stadt)
```

lauten.

Ein Expertensystem wie das medizinische Diagnose-System verwendet normalerweise zwei Datenbank-Prädikate, eins für Fakten, die als wahr, und eins für Fakten, die als falsch befunden wurden:

```
xpositiv(symptom)
xnegativ(symptom)
```

In unserem Diagnose-System wird das Prädikat *xpositiv(symptom)* zum Speichern der wahren Fakten und das Prädikat *xnegativ(symptom)* zum Speichern der falschen Fakten gebraucht. Damit können die Fakten, die aus den Antworten des Benutzers hervorgehen, in der Datenbank gespeichert werden, und die Datenbank kann befragt werden, bevor die gleiche Frage noch einmal gestellt würde. Ein Prädikat befindet sich entweder im Abschnitt *database* oder *predicates*, auf keinen Fall in beiden gleichzeitig. Betrachten Sie einmal das folgende Beispiel:

```
domains
    name,strasse,postleitzahl,stadt = string

database
    adresse(name,strasse,postleitzahl,stadt)
```

Hierdurch wird *adresse* als Datenbank-Prädikat definiert. Die Komponenten *name, strasse, postleitzahl, stadt* werden im Abschnitt *domains* eingegeben.

Verwendung der dynamischen Datenbank

Datenbank-Prädikate werden genau wie andere Prädikate verwendet. Sie erhalten den Wahrheitswert *True*, wenn in der Datenbank eine Unifikation erreicht wird, und *False*, wenn das nicht der Fall ist. Neue Fakten werden in den Datenbank-Prädikaten mittels der Standard-Prädikate *asserta(Faktum)* und *assertz(Faktum)* gespeichert. Mit *retract(Faktum)* können sie wieder aus dem Prädikat entfernt werden. Wir wollen uns nun das medizinische Diagnose-Programm in Abb. 12.2 anschauen, um zu verstehen, wie die dynamische Datenbank arbeitet. In diesem Beispiel wurde zur Vereinfachung die Abfrage nach dem Namen des Patienten weggelassen.

Die *los*-Klausel ruft *annahme(Krankheit)* auf. Die erste zu prüfende Bedingung in dieser Regel ist *symptom(fieber)*. In der Regel *symptom(fieber)* lautet die erste Bedingung *positiv(Frage,Symptom)*. Die erste Klausel, die daraufhin mit dem Prädikat *positiv* unifiziert, ist

```
positiv(_,Symptom):-
    xpositiv(Symptom),!.
```

Das bedeutet, daß die erste *positiv*-Regel die Datenbank nach *xpositiv(fieber)* überprüft (der Patient hat Fieber). Dabei kann es zu keiner Unifikation kommen, da die Datenbank zur Zeit noch leer ist. Es setzt Backtracking zu der Klausel

```
positiv(Frage,Symptom):-
    not(xnegativ(Symptom)),
    frag(Frage,Symptom,Eingabe),
    Eingabe = 'j'.
```

ein.

Die zweite *positiv*-Klausel überprüft das zweite Datenbank-Prädikat *xnegativ(fieber)* (kein Fieber). Diese Überprüfung scheitert, aber durch das Prädikat *not* wird der Wahrheitswert des Ergebnisses umgekehrt.

Das Prädikat *frag(Frage,Symptom,Eingabe)* stellt dann die Frage an den Benutzer und ruft *erinnere(Symptom,'j')* oder *erinnere(Symptom,'n')* auf. Diese Regeln speichern die Antwort mit dem Standard-Prädikat *asserta(Faktum)* in einem der beiden Datenbank-Prädikate.

Der Einsatz des Schnitts in der ersten Klausel *positiv(_,Symptom)* verhindert das Backtracking und bindet damit das Programm an diese Klausel, wenn sie erfolgreich ist (es handelt sich hier um einen grünen Schnitt – siehe Kapitel 9). Wenn beispielsweise der Patient Fieber, aber keinen Husten hat, wird die erste Regel *annahme(masern)* scheitern und Backtracking zu *annahme(röteln)* einsetzen.

Da der Patient Fieber hat, wird nun die Klausel *positiv(_,Symptom)* erfolgreich überprüft. Wenn von da ab irgendeine der folgenden Bedingungen der Regel *annahme(röteln)* scheitert, dann verhindert der Schnitt das Backtracking des Programms und somit die Überprüfung der Klausel *positiv(Frage,Symptom)*.

Im folgenden sind die wesentlichen Punkte für das Speichern von Daten in einer Datenbank zusammengefaßt:

1. Das Prädikat *asserta(Faktum)* speichert ein Faktum am Anfang der Datenbank, das Prädikat *assertz(Faktum)* am Ende. Welches von beiden für Sie in Frage kommt, hängt davon ab, wo Sie das Faktum in der Datenbank brauchen. Um das Programm leistungsfähig zu machen, sollten Sie die Position nach der Häufigkeit wählen, mit der auf das Faktum zugegriffen werden soll.

2. Alle Variablen in diesem Faktum müssen gebunden sein, bevor die Datenbank-Prädikate *asserta(Faktum)* oder *assertz(Faktum)* aufgerufen werden.

3. Das Faktum in den Standard-Datenbank-Prädikaten kann jeder beliebige Ausdruck eines Faktums sein und auch mehrere Argumente enthalten. Um beispielsweise im medizinischen Diagnose-System den Namen des Patienten zusammen mit dessen Symptom zu speichern, könnte man das Datenbank-Prädikat wie folgt umdefinieren:

```
asserta(xpositiv(name,symptom))
```

Löschen von Fakten

Ein Faktum, das in der Datenbank gespeichert ist, kann nur mit Hilfe des Standard-Prädikats *retract(Faktum)* wieder gelöscht werden. Zum Beispiel entfernt

```
retract(xpositiv(fieber))
```

das Faktum *fieber* aus der Datenbank *xpositiv(symptom)*. Variablen müssen nicht unbedingt vor dem Aufruf des Prädikats gebunden sein. Beispielsweise wird durch

```
retract(xpositiv(_))
```

das erste Faktum, das mit der angegebenen Bedingung unifiziert, aus der Datenbank entfernt.

In unserem Diagnose-System muß die Datenbank gelöscht werden, bevor die Daten für den nächsten Patienten eingegeben werden. Das wird mit dem Prädikat *lösch_fakten* bewerkstelligt, und zwar wie folgt:

```
lösch_fakten:-
    retract(xpositiv(_)),fail.

lösch_fakten:-
    retract(xnegativ(_)),fail.

lösch_fakten.
```

Dieser Programmteil enthält zwei Schleifen. Die erste Klausel

```
lösch_fakten:-
    retract(xpositiv(_)),fail.
```

löscht ein Element aus der Datenbank und scheitert dann aufgrund des Prädikats *fail*. Es setzt Backtracking zum *retract*-Prädikat ein, das daraufhin ein weiteres Element aus der Datenbank entfernt. Wenn die Datenbank leer ist, erfolgt das Backtracking zur zweiten Klausel:

```
lösch_fakten:-
    retract(xnegativ(_)),fail.
```

Diese zweite Klausel stellt ebenfalls eine Schleife dar, in der das zweite Datenbank-Prädikat gelöscht wird. Sobald auch dieses leer ist, wird das Prädikat *retract* scheitern, und das Backtracking erfolgt zur dritten Klausel, die immer erfolgreich ist.

Um in einer Datenbank ein Faktum zu ändern, müssen Sie es zunächst löschen und dann mit den Änderungen neu eingeben. Es gibt kein Standard-Prädikat, mit dem Fakten in einer Datenbank editiert werden können.

Speichern der dynamischen Datenbank

In unserem medizinischen Diagnose-System besteht die Datenbank nur für eine einzige Befragung. Sie wird gelöscht, wenn das Programm erneut durch-

laufen wird. Es kann jedoch manchmal wünschenswert sein, die dynamische Datenbank zu speichern und erneut zu verwenden. Das kann zum Beispiel bei einer Datenbank mit Adressen der Fall sein.

Diese Adressenliste wird mittels eines Programms regelmäßig auf den neuesten Stand gebracht, und das Programm soll nicht nach jeder Adressenänderung neu compiliert werden. In diesem Fall wird man die Adressen in einer dynamischen Datenbank speichern und diese in einer Datei bis zur nächsten Bearbeitung ablegen.

Zum Speichern der aktuellen dynamischen Datenbank dient das Standard-Prädikat *save(dateiname)*, bei dem der Dateiname und eine Erweiterung wie folgt angegeben werden muß:

```
save("adressen.dat")
```

Dieses Prädikat hat immer den Wahrheitswert *True*. Die entstehende Datei besteht aus einer Sammlung von Fakten und sieht wie jedes andere Prolog-Programm aus, nur daß sie keine Abschnitt-Bezeichnungen führt. Jederzeit danach können Sie von diesem (oder einem anderen) Programm aus diese dynamische Datenbank unter Verwendung des Prädikats *consult(dateiname)* wieder einlesen:

```
consult('adressen.dat')
```

Dieses Prädikat besitzt ebenfalls den Wahrheitswert *True*. Sobald die Fakten wieder in den Rechnerspeicher eingelesen wurden, sind sie wieder ein Teil des aktuellen Programms. Achten Sie immer darauf, daß genügend Speicherplatz für die Klauseln Ihres Programms und die dynamische Datenbank vorhanden ist.

Erstellen von Menüstrukturen

Mit dem, was Sie in den letzten Kapiteln gelernt haben, können Sie bereits eine einfache Menüstruktur für das medizinische Diagnose-System erstellen. Abb. 12.3 zeigt einen Teil dieses Menüs. Die erste Klausel

```
prüfsymptom(Symptom,X):-
    xpositiv(Symptom,Y),X = Y,.
```

prüft, ob das Menü bereits angezeigt wurde und ob bereits irgendeine Auswahl für das betreffende Symptom aus der vorhergegangenen Menüauswahl in der

Datenbank gespeichert worden ist. Zu diesem Zweck wird die Variable *Symptom* gebunden und nach einer Unifikation mit *xpositiv(Symptom,_)* in der Datenbank gesucht.

```
prüfsymptom(Symptom,X):-
   xpositiv(Symptom,Y),
   X =Y,!.

prüfsymptom(fieber,X):-
   not(xpositiv(Symptom,_)),
   repeat,
   write("Wie ist das Fieber?"),nl,
   write("1. gleichbleibend hoch"),nl,
   write("2. leicht"),nl
   write("3. mittel"),nl,
   write("4. hoch und schnell ansteigend"),nl,
   write("AUSWAHL: "),
   antwort(Eingabe),
   char_int(Eingabe,I),
   I<53, I> 48,
   asserta(xpositiv(fieber,Eingabe)),
   X = Eingabe.
```

Abb. 12.3: Ein einfaches Menüsystem

Die Klausel

```
X = Eingabe
```

wird in diesem Fall, wenn ein Menüwert Y für das Symptom gespeichert wurde, erfolgreich sein. Erfolg oder Scheitern der Klausel hängt davon ab, ob der gespeicherte Wert mit X im Regelkopf unifiziert oder nicht.

Wenn in der Datenbank kein Eintrag für das Symptom zu finden ist, kommt die nächste Klausel zur Ausführung, die dem Anwender ein Menü präsentiert. Dabei ermöglicht die repeat-Schleife Backtracking, falls der Anwender eine falsche Eingabe getätigt hat.

Der Eingabewert wird in der Datenbank gespeichert. Die Überprüfung dieser Klausel ist erfolgreich, wenn die Menüauswahl dem Wert entspricht, an den X im Regelkopf gebunden ist.

Beachten Sie auch, wie der Eingabewert in eine ganze Zahl umgewandelt und seine Zulässigkeit geprüft wird. (Die ASCII-Werte für die Zahlen 1, 2, 3 und 4 sind 49, 50, 51 und 52.) Wenn ein nicht erlaubter Wert eingegeben wird, wartet das Programm nach Backtracking zu repeat auf eine neue Eingabe.

Übungen

1. Ändern Sie das Programm aus Abb. 12.2 so ab, daß der Name des Patienten berücksichtigt und zusammen mit dessen Symptom in der dynamischen Datenbank abgelegt wird.

2. Nehmen Sie an, daß ein Patient unter Fieber, Kopfschmerzen, Schnupfen und Hautausschlag leidet. Beschreiben Sie den Inhalt der dynamischen Datenbank nach Beendigung des Programmlaufs. Überprüfen Sie Ihre Vermutung, indem Sie die dynamische Datenbank mit Hilfe des entsprechenden Prädikats abspeichern.

3. Schreiben Sie ein Prolog-Programm, das eine Datenbank mit Adressen (Name, Straße, Postleitzahl und Stadt) erstellt, und speichern Sie diese als dynamische Datenbank für weitere Arbeiten ab. Verwenden Sie das Prädikat *consult*, um die Datenbank nach dem Programmstart in das Programm einzulesen, und das Prädikat *save*, um sie vor dem Programmende abzuspeichern.

Kapitel 13

Arbeiten mit Listen

Eine der leistungsfähigsten Strukturen von Prolog ist die Liste. Die Möglichkeit, Probleme mit Hilfe von Listen zu lösen, ist einer der Hauptvorteile, die Prolog gegenüber anderen Sprachen, wie z. B. BASIC oder COBOL, bietet. In diesem Kapitel werden Sie sich davon überzeugen können.

Listen

Eine *Liste* ist eine geordnete Folge von Termen. Dabei spielt das Wort *geordnet* eine Schlüsselrolle, da es wesentlich auf die Reihenfolge der Terme ankommt. Listenterme können Variablen, einfache oder zusammengesetzte Objekte oder andere Listen sein. Eine Liste kann eine unbegrenzte Anzahl von Termen enthalten.

Jede Liste muß in eckige Klammern gesetzt werden, wobei die einzelnen Komponenten durch Kommas voneinander getrennt werden müssen:

```
[alfred,susanne,till,rita,frank,lisa]
```

Da es bei einer Liste, wie bereits erwähnt, auf die Reihenfolge ankommt, ist die vorige Liste nicht die gleiche wie

```
[alfred,rita,frank,lisa,susanne,till]
```

obwohl beide aus sechs gleichen Termen zusammengesetzt sind.

Die Komponenten einer Liste sollten alle vom gleichen Domain-Typ sein. Sie können ganze Zahlen, Dezimalzahlen, Objekte vom Typ *symbol*, Zeichenketten, zusammengesetzte Objekte oder selbst wieder Listen sein. Wir werden uns in diesem Buch bei der Erläuterung allgemeiner Aussagen der Schreibweise

```
[a,b,c,d]
```

bedienen. Eine leere Liste (eine Liste ohne Elemente) wird durch das leere Klammernpaar

```
[ ]
```

angezeigt. Ihr Einsatz ist bei der Überprüfung von Listen-Prädikaten sinnvoll, wie wir später noch sehen werden.

Deklaration von Listen

Prädikate können Listen genau wie alle anderen Argumente verwenden:

```
namen([alfred,susanne,till,rita,frank,lisa])
```

Ein Prädikat mit mehreren Argumenten, die alle Listen sind, wird genauso geschrieben:

```
autos([farben],[fabrikate],[baujahre])
```

Eine Liste erhält ihre Typzuweisung als Liste im Abschnitt *domains*:

```
domains
    namenliste = namen*
    namen = symbol

predicates
    club_namen(namenliste)

clauses
    club_namen([alfred,susanne,till,rita,frank,lisa]).
```

Die folgenden Punkte fassen die wesentlichen Merkmale von Listen in Turbo-Prolog zusammen:

- Eine Liste wird durch einen Stern * bei der Deklaration gekennzeichnet. Der Stern besagt, daß es mehrere Listenelemente (auch Null, d.h eine leere Liste mit keinem Element ist zulässig) geben kann. Mit dem Stern wird der Name des Listenelements gekennzeichnet. Diesem Elementnamen wird in einer weiteren Anweisung ein Typ zugewiesen. In verkürzter Schreibweise kann dieser Typ selbst als Elementname verwendet werden. Für das vorige Beispiel muß es dann heißen:

  ```
  namenliste = symbol*
  ```

- Alle Objekte einer Liste müssen vom gleichen Typ sein, diese können aber so komplex wie nötig sein.

- Es können Prädikate definiert werden, die den Listennamen als Argument verwenden.

Unifikation bei Listen

Um Listen auf Unifikation zu prüfen, teilt Prolog jede Liste in einen Kopf und eine Restliste. Der Kopf ist das erste Element der Liste, und die Restliste

enthält die restlichen Terme. In der Namensliste beispielsweise ist *alfred* der Kopf, und die Restliste lautet *[susanne,till,rita,frank,lisa]*. In Listen kann auch der senkrechte Doppelstrich (|) benutzt werden, um den Kopf von der Restliste zu trennen:

```
[a|[b,c,d]]
```

Im Namenbeispiel hieße es dann:

```
[alfred|[susanne,till,rita,frank,lisa]]
```

In allgemeiner Form kann es ausgedrückt werden als:

```
club_name([Kopf|Restliste])
```

Beim Unifikationsprozeß wird Prolog die beiden Strukturen

```
[Kopf|Restliste] = [a,b,c,d]
```

durch Binden von *Kopf* an *a* und *Restliste* an *[b,c,d]* unifizieren.

Mit dieser Schreibweise kann das einzige Faktum in unserem Beispiel geschrieben werden als:

```
club_namen([alfred,susanne,till,rita,frank,lisa]).
```

oder

```
club_namen([alfred|[susanne,till,rita,frank,lisa]]).
```

oder

```
club_namen([alfred|[susanne|[till,rita,frank,lisa]]]).
```

Alle drei Ausdrücke enthalten das gleiche Faktum und können gleichwertig benutzt werden. Um zu verstehen, wie Turbo Prolog Listen unifiziert, können Sie das Programm aus Abb. 13.1 eingeben und compilieren. Bei der Ausführung des Programms geben Sie das Ziel

```
Goal: club_namen(A)
```

ein. Sie sehen eine Liste:

```
A=["alfred","susanne","till","rita","frank","lisa"]
1 Solution
```

Versuchen Sie es einmal mit folgendem Ziel:

```
Goal: club_namen([A|B])
```

Dieses Mal lautet die Antwort:

```
A=alfred,   B=["susanne","till","rita","frank","lisa"]
1 Solution
```

In diesem Fall unifizierte A mit dem Kopf der Liste und wurde an ("alfred") gebunden. B unifizierte mit der Restliste und wurde dementsprechend an (["susanne","till","rita","frank","lisa"]) gebunden. Beachten Sie, daß sowohl der Kopf als auch die Restliste von Typ *symbol* ist.

```
domains
   namensliste = namen*
   namen = symbol

predicates
   club_namen(namenliste)

clauses
   club_namen([alfred,susanne,till,rita,frank,lisa]).
```

Abb. 13.1: Einfaches Listenbeispiel

Versuchen Sie abschließend einmal das folgende Ziel:

```
Goal: club_namen([A,B|C])
```

Die Lösung lautet:

```
A=alfred, B=susanne, C=["till","rita","frank","lisa"]
```

Hierbei unifiziert A mit dem ersten Element der Liste, B mit dem Kopf der Restliste und C mit dem Rest der Restliste.

Verändern Sie die Klausel im Programm einmal wie folgt:

```
club_namen([alfred|[susanne,till,rita,frank,lisa]]).
```

Auch jetzt erhalten sie wieder die gleichen Antworten. Nun ändern Sie die Klausel erneut:

```
club_namen([alfred|[susanne|[till,rita,frank,lisa]]]).
```

Überprüfen sie auch diesmal wieder die Antworten. Alle drei Formen des Ausdrucks sollten zu dem gleichen Ergebnis führen.

Das Beispiel war recht einfach, aber es sollte ausreichen, Ihnen einen ersten Einblick darin zu geben, wie in Turbo Prolog Listen unifiziert werden. Probieren Sie noch ein paar Beispiele aus, bis Sie sicher sind, daß Sie das Prinzip der Liste verstanden haben und wie ihre Komponenten – Kopf und Restliste – unifiziert werden können. Versuchen Sie zum Abschluß noch die Eingabe des Ziels

```
Goal: club_namen([A,B,C,D,E,F|G])
```

Dieses Mal lautet die Antwort:

```
A=alfred, B=susanne, C=till, D=rita, E=frank,
F=lisa, G=[]
```

Vielleicht haben Sie auch bemerkt, daß Turbo Prolog mehr Zeit für die Ausgabe dieser Antwort gebraucht hat.

Listen von Listen

Prolog-Strukturen können so komplex wie nötig gemacht werden. Dabei können Sie Listen innerhalb von Listen verwenden, so wie Sie zusammengesetzte Objekte innerhalb von zusammengesetzten Objekten verwenden können. Ein Beispiel für eine Liste mit Unterlisten könnte so aussehen:

```
lebewesen([tiere,pflanzen])
tiere([hund,katze,maus])
pflanzen([kaktus,baum,gras])
```

Verwendung von zusammengesetzten Objekten

Sie können Listen mit Unterlisten auch unter Verwendung von zusammengesetzten Objekten definieren. Das vorige Beispiel kann dann wie folgt geschrieben werden:

```
lebewesen(tiere([hund,katze,maus]),
pflanzen([kaktus,baum,gras]))
```

Die Listenstruktur in Prolog kann nur bedingt mit Feldern (arrays) in prozeduralen Programmiersprachen verglichen werden. Felder haben eine feste Länge und Dimension, während Listen von variabler Länge und Verschachtelungstiefe sind und eine hierarchische Struktur darstellen können. Bei Feldern kann die Adresse der Komponenten ganz einfach berechnet werden. Eine Komponente einer Liste kann nur angesprochen werden, wenn die Komponenten der Liste der Reihe nach bis zur gewünschten durchgegangen werden.

Programmieren mit Listen

Beim Programmieren mit Listen wird ständig mit Rekursionen gearbeitet. Dabei kann das Prädikat *mitglied* eingesetzt werden, wenn man herausfinden will, ob eine Komponente Bestandteil der Liste ist:

```
mitglied(X,[X|_]).
mitglied(X,[_|Rest]):-
   mitglied(X,Rest).
```

Das Programm in Abb. 13.2 verwendet dieses Prädikat. Tippen Sie dieses Programm ein, und compilieren Sie es. Als Ziel geben Sie

```
Goal: club_mitglied(susanne)
```

ein, und Turbo Prolog wird antworten:

```
True
```

Die erste Klausel mit dem *mitglied*-Prädikat ist erfolgreich, wenn das Wort *susanne* den Kopf der Liste bildet. Da aber *alfred* der Kopf ist, scheitert die Klausel. Nun wird die zweite Klausel mit dem *mitglied*-Prädikat überprüft. Diese Klausel bedient sich der Rekursion, um herauszufinden, ob *susanne* ein Element der Restliste ist. Zunächst wird geprüft, ob *susanne* der Kopf der Restliste ist. Diese Klausel ist bereits erfolgreich.

```
domains
   namenliste = symbol*

predicates
   mitglied(symbol, namenliste)
   club_mitglied(symbol)

clauses
   mitglied(X,[X|_]).

   mitglied(X,[_|Rest]):-
      mitglied(X,Rest).

   club_mitglied(Name):-
      mitglied(Name,[alfred,susanne,till,rita,frank,lisa]).
```

Abb. 13.2: Beispielprogramm für das Prädikat mitglied

Wenn die Prädikat-Struktur von *mitglied* das gesuchte Element nicht im Listenkopf findet, durchsucht es die Restliste mittels Rekursion und prüft jede Komponente auf den eingegebenen Namen hin.

Geben Sie jetzt das Ziel

```
Goal: club_mitglied(lore)
```

ein, und Turbo Prolog antwortet:

```
False
```

Zum Schluß sollten Sie noch das Ziel

```
Goal: club_mitglied(X)
```

versuchen. Dieses Mal gibt Prolog alle Werte von X aus, die mit dem Ziel unifizieren; d. h. die Namen aller Mitglieder werden ausgegeben.

Listenoperationen

Es gibt einige Prädikate für Listenoperationen, die Sie definieren und zum festen Bestandteil Ihrer Prolog-Library machen sollten. Fast alle arbeiten rekursiv, wie auch das Prädikat *mitglied*. Wir wollen im folgenden einige nützliche Prädikate näher beschreiben.

Schreiben von Listen

Listen können nicht mit dem Standard-Prädikat *write* geschrieben werden. Wir definieren deshalb das Prädikat *schreibliste*, das eine Rekursion verwendet. Wie das Prädikat *mitglied* ist auch dieses nicht standardmäßig verfügbar. Sein Format ist:

```
schreibliste([ ]).

schreibliste([Kopf|Rest]):-
    write(Kopf),nl,schreibliste(Rest).
```

Das Prädikat *schreibliste* arbeitet eine Liste ab und gibt dabei jedes Mal den Kopf der noch verbleibenden Restliste aus:

```
Goal: schreibliste([a,b,c])
a
b
c
True
```

Verketten von Listen

Um zwei Listen miteinander zu verbinden, können Sie das Prädikat *append* definieren (was in prozeduralen Sprachen einem gleichlautenden Befehl für das Aneinanderhängen von Dateien entspricht). Sein Format lautet:

```
append([ ],Liste,Liste).
```

```
append([X|Liste1],Liste2,[X|Liste3]):-
    append(Liste1,Liste2,Liste3).
```

Das folgende Beispiel verwendet das append-Prädikat:

```
Goal: append([a,b,c],[d,e],X)
X=["a","b","c","d","e"]
1 Solution
```

Das Prädikat kann auch umgekehrt dazu dienen, die zugehörige Eingabeliste zu finden, wenn die zweite Eingabeliste und die Ausgabeliste bekannt sind:

```
Goal: append(X,[d,e],[a,b,c,d,e])
X=["a","b","c"]
1 Solution
```

Umkehren einer Liste

Das Prädikat *umkehrliste* kehrt die Reihenfolge der Elemente einer Liste um. Sein Format lautet:

```
umkehrliste(Eingabeliste,Ausgabeliste):-
    umkehr(Eingabeliste,[],Ausgabeliste).
    umkehr([],Eingabeliste,Eingabeliste).

umkehr([Kopf|Rest],Liste1,Liste2):-
    umkehr(Rest,[Kopf|Liste1],Liste2).
```

Ein Beispiel für die Verwendung des Prädikats *umkehrliste* ist:

```
Goal: umkehrliste([a,b,c,d],X)
X=["d","c","b","a"]
1 Solution
```

Bestimmung des letzten Elements einer Liste

Das Prädikat *letztes_element* findet das letzte Elemente in einer Liste heraus. Das Format lautet:

```
letztes_element([Kopf],X):-
    X = Kopf.

letztes_element([_|Rest],X):-
    letztes_element(Rest,X).
```

Der Programmlauf zeigt:

```
Goal: letztes_element([a,b,c],X)
X=c
1 Solution
```

Bestimmung des n-ten Elements einer Liste

Das Prädikat *n_element* findet das n-te Element in einer Liste heraus. Das Format lautet:

```
n_element([Kopf|_],1,X):-
    X = Kopf.

n_element([_|Rest],N,X).

n_element([Kopf|Rest],N,X):-
    NN=N - 1,
    n_element(Rest,NN,X).
```

Ein Beispiel für die Verwendung des Prädikats *n_element* ist:

```
Goal: n_element([a,b,c,d],3,X)
X=c
1 Solution
```

Verschiedene Domain-Typen in einer Liste

Wenn Sie eine Liste benutzen wollen, die mehrere Typen enthalten soll, so müssen Sie die Elemente im Abschnitt *domains* wie folgt definieren:

```
domains
    namen = str(string);int(integer);ch(char)
    namenliste = namen*

predicates
    mitglied(namen,namenliste)
```

Innerhalb einer Liste können Sie nicht verschiedene Typen mischen, aber Sie können die gleiche Liste mit mehr als einem Typ verwenden. So kann beispielsweise *namenliste* im obigen Beispiel aus Elementen vom Typ *string*, *integer* oder *char* bestehen.

Übungen

1. Bestimmen Sie jeweils den Kopf und den Rest der folgenden Listen.

a. [2,56,1,9,3,41,87]

b. [rot,blau,schwarz,orange]

c. [albert,franz,lilli,rosi]

d. [ ]

2. Ändern Sie das Beispielprogramm zur medizinischen Diagnose aus Kapitel 12 so ab, daß die Reihenfolge der getesteten Annahmen über eine Liste gesteuert wird. Diese Liste sol so gebildet sein, daß die Krankheiten in der folgenden Reihenfolge geprüft werden: *erkältung, grippe, masern, röteln, mumps, windpocken, keuchhusten.*

Kapitel 14

Zeichenkettenoperationen

In Turbo Prolog gibt es mehrere Standard-Prädikate für die Verarbeitung von Zeichenketten. In diesem Kapitel werden wir die Prädikate *concat, frontstr, str_len, frontchar, fronttoken, isname* und *upper_lower* kennenlernen.

Zeichenketten in Turbo Prolog

Wie wir bereits wissen, gibt es für Objekte in Turbo Prolog sechs verschiedene Typen: *char, string, real, integer, symbol* und *file*. Zeichenketten sind vom Typ *string* und können eine beliebige Zeichenfolge darstellen, die in Anführungszeichen eingeschlossen werden muß, wie z. B.:

"Herr Wagner"

"123"

"&abc"

Eine Zeichenkette ist im Grunde genommen eine Liste von einzelnen Zeichen. Dazu ist folgendes zu bemerken:

- Alle Zeichen in einer Zeichenkette müssen vom gleichen Typ sein.
- Eine Zeichenkette ist eine Liste von ganz bestimmten Zeichen."Peter" ist deshalb nicht das gleiche wie "peter".
- Die Reihenfolge der Zeichen in einer Zeichenkette ist von Bedeutung. "XYZ" ist nicht die gleiche Zeichenkette wie "YXZ".
- Zeichenketten können beliebig lang sein.
- Zeichenketten können auch leer sein. Solche leeren Zeichenketten werden als ein Paar von Anführungszeichen geschrieben, z. B.:

```
write("").
```

Das Prädikat *write* wird ausgeführt, zeigt aber keine Wirkung. Eine leere Zeichenkette ist nicht das gleiche wie eine Zeichenkette, die ein Leerzeichen enthält. Eine Zeichenkette mit einem Leerzeichen würde ein Leerzeichen auf dem Bildschirm ausgeben.

Zeichenkettenoperationen

Mit Hilfe der Prädikate für Zeichenoperationen können Zeichenketten verkettet und Teilzeichenketten ermittelt werden. Dieselben Prädikate werden auch bei der Überprüfung von Beziehungen zwischen Zeichenketten eingesetzt.

Verketten

Das Standard-Prädikat *concat* verkettet zwei Zeichenketten zu einer dritten. Seine allgemeine Form ist:

```
concat(Zeichenkette1,Zeichenkette2,Zeichenkette3)
```

wobei die *Zeichenkette* das Ergebnis der Verketteung von *Zeichenkette1* und *Zeichenkette2* aufnimmt. Im Beispiel sieht das wie folgt aus:

```
los:-
X = "Morgenstund hat ",
Y = "Gold im Mund.",
concat (X,Y,Ausagbe),
write(Ausgabe).
```

Das Programm gibt den Satz aus:

```
Morgenstund hat Gold im Mund
```

Das Prädikat *concat* kann auch dazu verwendet werden, eine Teilzeichenkette zu liefern, wenn die Ergebniszeichenkette und eine der Eingabenzeichenketten bekannt ist. Beispielsweise führt das Programm

```
los:-
Z = "Morgenstund hat Gold ",
X = "Morgenstund",
concat(X,Y,Z),
write(Y).
```

zu der Ausgabe:

```
hat Gold
```

Das Prädikat ist deterministisch, d.h. mindestens zwei der Argumente müssen an Werte gebunden sein, und dann gibt es nur eine einzige mögliche Lösung.

Das Prädikat kann zu Testzwecken eingesetzt werden, indem alle drei Argumente gebunden werden. Wenn das der Fall ist, liefert das Prädikat den Wert *True*, wenn das dritte Argument der Summe der beiden ersten Argumente entspricht. Andernfalls wird der Wert *False* zurückgegeben.

Das Prädikat *frontstr*

Das Prädikat *frontstr* trennt eine vorgegebene Anzahl von Zeichen am Anfang der Zeichenkette ab. Die allgemeine Form lautet:

```
frontstr(Anzahl,Startzeichenkette,
    Ergebniszeichenkette,Restzeichenkette)
```

Dabei haben die einzelnen Argumente die folgenden Bedeutungen:

- Anzahl gibt die Anzahl der Zeichen an, die links abgetrennt werden sollen (Typ *integer*).
- Startzeichenkette ist die Eingabezeichenkette (Typ *string*).
- Ergebniszeichenkette nimmt die Zeichen auf, die links abgetrennt wurden (Typ *string*).
- Restzeichenkette beinhaltet den Rest der Eingabezeichenkette nach dem Abtrennen (Typ *string*).

Als Beispiel betrachten wir:

```
los:-
    X ="Morgenstund hat",
    frontstr(11,X,Ergebnis,Rest),
    write(Ergebnis),nl,
    write(Rest),nl.
```

Die Ausgabe lautet:

```
Morgenstund hat
```

Das Prädikat *frontstr* ist ebenfalls deterministisch. Die beiden ersten Argumente müssen gebunden und die anderen beiden Argumente frei sein. Das *frontstr*-Prädikat hat eine ähnliche Wirkung wie die LEFT$-Funktion in BASIC.

Im Gegensatz zum *concat*-Prädikat kann das *fronstr*-Prädikat nicht in umgekehrter Weise zum Verketten von Zeichenketten verwendet werden.

Das Prädikat *str_len*

Das Prädikat *str_len* liefert die Länge einer Zeichenkette. Seine allgemeine Form lautet:

```
str_len(Zeichenkette,Länge)
```

Dabei haben die einzelnen Argumente die folgenden Bedeutungen:

- Die Zeichenkette ist die Eingabezeichenkette, deren Länge bestimmt werden soll (Typ string).
- Die Länge gibt die Zeichenanzahl der Zeichenkette an.

Das erste Argument muß gebunden sein. Wenn das Längenargument frei ist, gibt das Prädikat in ihm die Länge der Zeichenkette zurück. Wenn das Längenargument bereits an einen Wert gebunden ist, dann hat das Prädikat eine Testfunktion und liefert den Wert *True*, wenn der Wert mit der Zeichenkettenlänge übereinstimmt, andernfalls wird der Wert *False* zurückgegeben.

Das Prädikat kann beispielsweise angewendet werden, um eine Zeichenkette rechtsbündig in ein Feld zu setzen:

```
rechtsbündig(Zeichenkette):-
    str_len(Zeichenkette,N),
    leerzeichen_links(N),
    write(Zeichenkette),nl.

leerzeichen_links(N):-
    X = 80 - N,
    leerzeichen(X).

leerzeichen(0):-.

leerzeichen(N):-
    write(" "),
    NN = N - 1,
    leerzeichen(NN).
```

Das Prädikat *frontchar*

Das Prädikat *frontchar* erfüllt eine ähnliche Funktion wie das Prädikat *frontstr*. Der Unterschied besteht darin, daß *frontchar* nur das erste Zeichen der Eingabezeichenkette liefert oder prüft. Seine allgemeine Form ist:

```
frontchar(Eingabezeichenkette,Zeichen,Restzeichenkette)
```

Dabei haben die einzelnen Argumente die folgenden Bedeutungen:

- Die Eingabezeichenkette ist die eingegebene Zeichenkette (Typ *string*).
- Das Zeichen ist das erste Zeichen der Eingabezeichenkette (Typ *char*).
- Die Restzeichenkette enthält die Eingabezeichenkette bis auf das erste Zeichen (Typ *string*).

Wenn das Zeichen gebunden ist, kann das Prädikat nur erfolgreich sein, wenn der Wert des Zeichens dem ersten Zeichen der Eingabezeichenkette entspricht. Wenn das Zeichen frei ist, wird es an den Wert des ersten Zeichens der Eingabezeichenkette gebunden. Mit diesem Prädikat kann von einer Zeichenkette eine Liste von Zeichen erzeugt werden oder umgekehrt.

Das Prädikat *fronttoken*

Das Prädikat *fronttoken* ermöglicht das Abtrennen eines Token vom Anfang einer Zeichenkette. Ein *Token* kann ein Name, eine Zahl oder ein beliebiges Zeichen außer dem Leerzeichen sein. In einem Satz ist jedes Wort ein Token. Mit dem Prädikat *fronttoken* kann z. B. ein Satz zergliedert werden, indem Wort für Wort abgetrennt wird.

Die allgemeine Form des Prädikats *fronttoken* lautet:

```
fronttoken(Eingabezeichenkette,Token,Restzeichenkette)
```

Dabei haben die einzelnen Argumente die folgenden Bedeutungen:

- Die Eingabezeichenkette ist die eingegebene Zeichenkette (Typ *string*).
- Das Token ist das erste Token in der Eingabezeichenkette (Typ *string*, *symbol*, ...).

- Die Restzeichenkette nimmt den Rest der Eingabezeichenkette ohne das erste Token auf (Typ *string*).

Wir betrachten ein Beispielprogramm:

```
X = "Morgenstund hat ",
    fronttoken(X,Ergebnis,Rest),
    write(Ergebnis),nl,
    write(Rest),nl.
```

Die Ausgabe lautet:

```
Morgenstund
hat
```

Das erste Token besteht aus den Zeichen bis zum ersten Leerzeichen (Morgenstund). Entweder muß die Eingabezeichenkette gebunden sein, oder das Token und die Restzeichenkette müssen gebunden sein. Wenn alle Argumente gebunden sind, prüft das Prädikat, ob der Wert von Token mit dem ersten Token der Eingabezeichenkette übereinstimmt. In dem Fall ist das Prädikat erfolgreich.

Das Prädikat *fronttoken* kann z. B. verwendet werden, um eine Listenstruktur mit Namen aus einer Zeichenkette zu bilden, in der die Namen durch Leerzeichen voneinander getrennt sind.

Das zurückgegebene Token kann mit den Prädikaten *isname*, *str_int* oder *str_len* ausgewertet und überprüft werden. Das Prädikat *str_int* wandelt das Token in sein ganzzahliges Äquivalent um. Die Länge des Token kann mit dem Prädikat *str_len* ermittelt werden. Über das Prädikat *isname* werden Sie im nächsten Abschnitt mehr erfahren. Ein Anwendungsbeispiel für das Zerlegen eines Satzes mit dem Prädikat finden Sie in Kapitel 20.

Das Prädikat *isname*

Das Prädikat *isname* dient zur Überprüfung, ob eine Zeichenkette der Syntax von Turbo Prolog für Zeichenketten genügt. Der Name muß mit einem Buchstaben beginnen und kann im weiteren aus Buchstaben, Ziffern oder Unterstrichen bestehen. Die allgemeine Form des Prädikats lautet:

```
isname(Zeichenkette)
```

wobei die Zeichenkette überprüft werden soll. Das Argument muß gebunden sein, und das Prädikat gibt dann entweder den Wert *True* (wenn die Zeichenkette ein Name ist) oder *False* zurück.

Das Prädikat *upper_lower*

Das Prädikat *upper_lower* wird zur Umwandlung von Groß- in Kleinbuchstaben oder umgekehrt gebraucht. Die allgemeine Form des Prädikats lautet:

```
upper_lower(Großzeichenkette,Kleinzeichenkette)
```

Wenn die Zeichenkette mit Kleinbuchstaben gebunden und die Zeichenkette mit Großbuchstaben frei ist, liefert das Prädikat zu der Zeichenkette in Kleinbuchstaben das Äquivalent in Großbuchstaben. Umgekehrt kann auch die Zeichenkette mit Großbuchstaben gebunden sein, dann wird das Äquivalent in Kleinbuchstaben zurückgegeben. Wenn beide Argumente gebunden sind, prüft das Prädikat, ob das erste Argument in Großbuchstaben dem zweiten in Kleinbuchstaben entspricht.

Übungen

1. Konstruieren Sie das Prädikat *teilkette(Startpos,Länge,Eingabekette,Ausgabekette)*, das eine Teilzeichenkette *Ausgabekette* von der *Eingabekette* mit einer Länge von *Länge* Zeichen, ab *Startpos* zurückgibt.

2. Definieren Sie das Prädikat *strpos(Eingabekette,Teilkette,Startpos)*, das das erste Auftreten von *Teilkette* in *Eingabekette* im Argument *Startpos* zurückgibt.

3. Definieren Sie das Prädikat *zentrier*, um eine Zeichenkette in einem Feld von 80 Zeichen Länge zu zentrieren.

Kapitel 15

Fenster, Grafik und Sound

Eine Eigenschaft, die Turbo Prolog von anderen Prolog-Versionen unterscheidet, ist die umfassende Unterstützung von Fenstern, Grafik, Tonausgabe und Dateiverarbeitung. In diesem Kapitel wollen wir eine Einführung in die Technik von Fenstern, Grafik und Sound geben. Das folgende Kapitel wird Ihnen dann die Verwendung der Datei-Prädikate näherbringen.

Sie können Fenstertechnik auf jedem Monochrom-, CGA- oder EGA-Monitor anwenden. Grafik-Prädikate können nur bei CGA- oder EGA-Monitoren eingesetzt werden.

Erstellen von Fenstern

Sie können in Ihrem Programm jederzeit durch Verwendung des Prädikats *makewindow* ein Fenster erstellen. Fügen Sie das Prädikat versuchsweise in eins der einfachen Beispielprogramme ein, und zwar entweder als Teil eines zusammengesetzten Ziels oder als erste Bedingung der höchstrangigen Regel (in diesem Buch ist das meistens die Regel *los*):

```
los:-
    makewindow(1,7,7,'',0,0,25,80),
    write('Ich mag Fenster.'),
    readchar(_).
```

Compilieren Sie das Programm, und führen Sie es aus. Auf dem Bildschirm wird ein Rahmen gezeichnet, und in diesem Fenster erscheint der Ausgabetext. Die vier gewohnten Prolog-Fenster werden erst wieder sichtbar, wenn das Programm durch Drücken einer beliebigen Taste beendet wird.

Das allgemeine Format des Prädikats *makewindow* lautet:

```
makewindow(Fensternummer,Bildschirmattribut,Rahmenattribut,
 Titel,Zeile,Spalte,Höhe,Breite)
```

Bei der Erstellung eines Fensters müssen alle diese Argumente gebunden sein. Außer dem Argument *Titel* sind alle Argumente ganze Zahlen. Der Titel kann vom Typ *string* oder *symbol* sein.

Die Bedeutungen der einzelnen Argumente sind die folgenden:

- Die Fensternummer ordnet jedem Fenster eine Nummer zu. Mit dem Prädikat *shiftwindow* kann zwischen den einzelnen Fenstern umgeschaltet werden.
- Das Bildschirmattribut legt die Anzeigemerkmale für das Fenster fest (siehe Tabellen 15.1 und 15.2).
- Das Rahmenattribut legt die Anzeigemerkmale für den Rahmen fest (siehe Tabellen 15.1 und 15.2).
- Der Titel gibt dem Fenster eine Überschrift.
- Zeile und Spalte legen den Anfangspunkt (obere linke Ecke) des Fensters fest.
- Höhe und Breite definieren die Größe des Fensters.

Anzeige	Wert	Vordergrund	Hintergrund
Leer	0	Schwarz	Schwarz
Normal	7	Weiß	Schwarz
Invers	112	Schwarz	Weiß

Anmerkung: Die Bezeichnung *Vordergrund* bezieht sich auf Text oder Zeichen.

Um die Zeichen in der Vordergrundfarbe zu unterstreichen, zählen Sie 1 hinzu.

Um den weißen Teil des Bildschirms leuchtend darzustellen, zählen Sie 8 hinzu.

Um die Zeichen im Vordergrund blinkend darzustellen, zählen Sie 128 hinzu.

Tabelle 15.1: Attributwerte beim Monochrom-Monitor

Farbe	Vordergrund	Hintergrund
Schwarz	0	0
Blau	1	16
Grün	2	32
Türkis	3	48
Rot	4	64
Lila	5	80
Braun	6	96
Weiß	7	112
Grau	–	8
Hellblau	–	24
Hellgrün	–	40
Helltürkis	–	56
Hellrot	–	72
Rosa	–	88
Gelb	–	104
Leuchtendweiß	–	120

Anmerkung: Der Attributwert für eine gewünschte Kombination berechnet sich durch Addition des Vordergrundwertes mit dem Hintergrundwert.

Tabelle 15.2: Attributwerte beim Farb-Monitor

In der Zielaussage oder in der höchstrangigen Regel können mehrere Fenster definiert werden, denen zur Unterscheidung verschiedene Nummern zugeord-

net werden. Mit dem Prädikat *shiftwindow* kann zwischen den Fenstern hin- und hergeschaltet werden:

```
shiftwindow(Fensternummer)
```

Wenn Sie auf ein Fenster umschalten, ist dieses zunächst leer, und der Cursor befindet sich in der oberen linken Ecke. Wenn Sie das Fenster verlassen, speichert Prolog die aktuelle Cursorposition, und bei Rückkehr in dieses Fenster steht der Cursor wieder auf dieser Position.

Fenster können einander überlagern. Die folgenden Prädikate definieren zwei Fenster in sogenannter "Overlay"-Technik. Dabei ist der Rahmen des zweiten Fensters unsichtbar, da es mit dem gleichen Bildschirmattribut wie das größere Fenster versehen ist, nämlich 7, und sein Rahmenattribut auf unsichtbar, also 0, gesetzt wird:

```
makewindow(1,7,7,"TESTFENSTER",0,0,25,80),
makewindow(1,7,0,'',3,3,21,76),
write('Das ist ein Test.'),
readchar(_).
```

Variieren Sie in dem Beispiel die Argumentwerte, um zu sehen, wie sich das Aussehen der Fenster ändert. Sie werden sehen, daß das Programm auch bei unzulässigen Argumentwerten anstandslos compiliert wird und erst der Programmlauf zu einer Fehlermeldung führt.

Probieren Sie auch die folgenden Werte für das Rahmenattribut des zweiten Fensters aus:

0	kein Rahmen
7	normaler Rahmen
112	inverser Rahmen
135	blinkender Rahmen

Der Attributwert für den blinkenden Rahmen berechnet sich aus dem Wert für das normale Attribut, z. B. 7, plus der Zahl 128, also insgesamt 135. Vordergrundzeichen können unterstrichen werden, indem eine 1 zu dem Attributwert hinzugezählt wird. Der weiße Bereich des Bildschirms kann intensiv leuchtend dargestellt werden, indem der Attributwert um 8 erhöht wird.

Die Prädikate zum Lesen und Schreiben beziehen sich jeweils auf das gerade aktive Fenster.

Allgemein ist die Verarbeitungsfolge bei Fenstern die folgende:

1. Alle Fenster, die in einem Programm benötigt werden, werden in der höchstrangigen Regel definiert. Dabei erhält jedes Fenster eine eigene Nummer.
2. Das Einschalten des ersten Fensters erfolgt mit dem Prädikat *shiftwindow*.
3. Während des Programmlaufs dient das Prädikat *shiftwindow* zum Umschalten zwischen den verschiedenen Fenstern.

Es gibt noch eine zweite Möglichkeit, das Prädikat *makewindow* zu verwenden. Wenn alle Argumente beim Aufruf des Prädikats frei sind, werden sie an die Werte des gerade aktiven Fensters gebunden.

Anwendung der Fenster-Technik

Für das Arbeiten mit Fenstern stehen mehrere unterstützende Prädikate zur Verfügung:

removewindow	entfernt das aktuelle Fenster vom Bildschirm.
clearwindow	löscht den Inhalt des aktuellen Fensters und positioniert den Cursor in die obere linke Ecke.
cursor(Zeile,Spalte)	setzt den Cursor im aktuellen Fenster auf die Position (Zeile,Spalte), wenn die Argumente gebunden sind. Wenn sie frei sind, wird die aktuelle Cursorposition zurückgegeben.
window_attr(Attribut)	ändert das Fensterattribut auf die durch das Argument angegebenen Werte, wenn das Argument gebunden ist. Wenn es frei ist, wird das aktuelle Fensterattribut zurückgegeben.
gotowindow(Fensternummer)	schaltet ähnlich wie *shiftwindow* zwischen verschiedenen Fenstern hin und her. Die Ausführung ist dabei schneller als bei *shiftwindow*, kann jedoch nicht bei überlappenden Fenstern angewendet werden. (Das Prädikat ist erst ab Version 1.1 verfügbar.)

Das Prädikat *cursor* setzt den Cursor in die angegebene Zeile und Spalte. Es kann z. B. zum Überschreiben von bereits auf dem Bildschirm stehenden Textzeilen verwendet werden. Normalerweise werden die Zeilen im Dialog-Fenster fortlaufen, eine nach der anderen, ausgegeben. Mit dem Prädikat *cursor* kann nun eine Zeile über einer anderen ausgegeben werden. Eine sinnvolle Anwendung dafür besteht in der Korrektur von fehlerhaften Eingaben und bei der Eingabe von aufeinanderfolgenden Zeilen gleichen Typs (z. B. mehrfache Adreßeingaben).

Das Prädikat *cursor* kann auch noch auf eine andere Art Verwendung finden. Wenn seine Argumente ungebunden sind, liefert es die aktuelle Cursorposition.

Das Prädikat *clearwindow* kann jederzeit im Programm eingesetzt werden, um den Inhalt eines Fensters zu löschen. Angenommen, Sie haben es mit einem Expertensystem zu tun, das menügesteuert arbeitet. Vor der Anzeige jedes neuen Menüs ist es sicher wünschenswert, das vorige zu löschen und vielleicht zwischendurch eine Überschrift auszugeben.

Bildschirmbezogene Ein- und Ausgabe

Wir haben bereits die einfachen Prädikate *read* und *write* zum Lesen und Schreiben von Daten kennengelernt. Manchmal ist jedoch eine anspruchsvollere Steuerung für Ein- und Ausgaben nötig. In Turbo Prolog gibt es deshalb eine umfassende Ein-/Ausgabeunterstützung auf drei Ebenen: für Zeichen, für Felder und für Fenster. In Turbo Prolog ist ein *Feld* eine zusammengehörende Folge von Zeichenpositionen.

Ein- und Ausgabe von Zeichen

Auf der Zeichenebene gibt es für die Ein- und Ausgabe zwei Prädikate. Mit dem Prädikat *scr_char* kann ein Zeichen an jeder beliebigen Stelle des Bildschirms gelesen oder geschrieben werden. Das Prädikat *scr_attr* ermöglicht es, das Attribut eines beliebigen Bildschirmzeichens zu setzen oder abzufragen. Das Format dieser Prädikate lautet:

```
scr_char(Zeile,Spalte,Zeichen)
scr_attr(Zeile,Spalte,Attribut)
```

Im ersten Prädikat müssen *Zeile* und Spalte an ganzzahlige Werte gebunden sein. Wenn *Zeichen* auch gebunden ist (Objekt vom Typ *char*), dann wird es

an der durch *Zeile* und *Spalte* festgelegten Position auf dem Bildschirm ausgegeben. Wenn *Zeichen* hingegen frei ist, so wird es durch durch das Prädikat an das Zeichen (auch an ein mögliches Leerzeichen) gebunden, das an der durch *Zeile* und *Spalte* festgelegten Position auf dem Bildschirm steht.

Auch beim zweiten Prädikat ist es erforderlich, daß *Zeile* und *Spalte* gebunden sind. Wenn *Attribut* ebenfalls gebunden ist, wird das Attribut für die durch *Zeile* und *Spalte* gekennzeichnete Position auf dem Bildschirm auf den Wert des neuen Attributs geändert.

Ist *Attribut* jedoch noch frei, so wird es an den Attributwert der durch *Zeile* und *Spalte* festgelegten Position gebunden. Das gibt Ihnen die Möglichkeit, beispielsweise bei der Verwendung von blinkenden Zeichen einzelne davon auszunehmen.

Ein- und Ausgabe von Feldern

Für die Ein- und Ausgabe oder das Setzen und Abfragen von Attributwerten einer Gruppe, die aus mehreren zusammenhängenden Zeichen besteht, sollten Sie die Prädikate für Feldoperationen benutzen. Diese heißen:

```
field_str(Zeile,Spalte,Länge,Zeichenkette)

field_attr(Zeile,Spalte,Länge,Attribut)
```

Jedes Prädikat muß zunächst das Feld definieren, bevor es Operationen mit ihm ausführen kann. Dazu dienen die ersten drei Argumente, die aus diesem Grund an ganzzahlige Werte gebunden sein müssen.

Das erste Prädikat gibt eine Zeichenkette aus oder liest sie ein. Wenn *Zeichenkette* gebunden ist, fungiert das Prädikat als Ausgabe-Prädikat und gibt den Inhalt der Zeichenkette an der durch *Zeile* und *Spalte* festgelegten Position aus.

Wenn das Argument *Zeichenkette* frei ist, dann ist das Prädikat ein Eingabe-Prädikat, das die Zeichenkette an den aktuellen Text bindet, der gerade im definierten Feldbereich angezeigt wird.

Mit dem zweiten Prädikat wird das Feldattribut verwaltet. Wenn das Argument *Attribut* gebunden ist, dann erhält das Feld dieses Attribut zugewiesen. Wenn *Attribut* noch frei ist, so wird es an den aktuellen Wert des Feldattributs gebunden.

Ein- und Ausgabe in Fenstern

Wie bei den Zeichen und Feldern gibt es auch für Fenster zwei Prädikate zur Ein- und Ausgabe:

```
window_str(Zeichenkette)

window_attr(Attribut)
```

Diese wirken jeweils nur im aktuellen Fenster.

Das Prädikat *window_str* kann eine oder mehrere Textzeilen in einem Fenster ausgeben oder den ganzen Inhalt eines Fensters lesen. Wenn *Zeichenkette* frei ist, fungiert das Prädikat wieder als Eingabe-Prädikat, und die Zeichenkette nimmt den Inhalt des Fensters auf. Dabei wird die Länge der Zeilen bis zum letzten Zeichen, das kein Leerzeichen ist, gerechnet.

Wenn *Zeichenkette* bereits gebunden ist, übernimmt das Prädikat die Funktion eines Ausgabe-Prädikats, das den Text in das aktuelle Fenster schreibt. Dabei gibt es keinen automatischen Wortumbruch. Wenn eine Zeile zu lang ist, wird sie abgeschnitten.

Das Prädikat *window_attr(Attribut)* (wir haben es bereits früher in diesem Kapitel schon einmal erwähnt) bindet das Argument *Attribut*, wenn es noch frei ist, an das Fensterattribut. Wenn *Attribut* gebunden ist, erhält das Fenster den gebundenen Wert als Attribut.

Verwendung des Editors im Programm

Das Editor-Fenster ist auch innerhalb von Programmen verfügbar. Für seine Handhabung sind drei Prädikate vorgesehen: *display, edit* und *editmsg*.

Das Prädikat *display*

Das Format des Prädikats *display* lautet:

```
display(Zeichenkette)
```

Das Prädikat gibt die Zeichenkette im aktuellen Fenster aus. Diese kann dann mit den Editor-Befehlen (siehe Kapitel 2) untersucht, aber nicht verändert werden. Das Argument *Zeichenkette* muß gebunden sein.

Das Prädikat *edit*

Das Format des Prädikats *edit* lautet:

```
edit(Eingabezeichenkette,Ausgabezeichenkette)
```

Das Argument *Eingabezeichenkette* muß gebunden und *Ausgabezeichenkette* muß frei sein. Beides sind Zeichenkettenwerte. Das Prädikat gibt die *Eingabezeichenkette* im aktuellen Fenster aus, wo es mit Hilfe der Editor-Befehle (siehe Kapitel 2) verändert werden kann. Dieses veränderte Ergebnis wird als *Ausgabezeichenkette* zurückgegeben.

Das Prädikat *editmsg*

Das Prädikat *editmsg* verfügt über acht Argumente, mit denen es zahlreiche Funktionen erfüllen kann: Aufruf des Editors, Ausgabe einer Zeichenkette im Editor-Fenster, Verändern der Zeichenkette, Text links oder rechts in der Titelzeile zufügen, Cursor steuern und Ausgabe einer Meldung am unteren Fensterrand. Mit *Titelzeile* ist hier der spezielle Titelbalken des Editor-Fensters gemeint. Sie können dieses Prädikat auch dazu verwenden, eine Datei mit Hilfsinformationen zu erstellen, die über die Funktionstaste <F1> erreicht werden kann. So könnten Sie für jedes Programm eine eigene Hilfsdatei bereitstellen.

Das Format des Prädikats *editmsg* lautet:

```
editmsg(Eingabezeichenkette,Ausgabezeichenkette,
Titellinks,Titelrechts,Meldung,Position,Dateiname,Code)
```

Die Argumente *Position* und *Code* sind vom Typ *integer*, die anderen vom Typ *string*. Beim Aufruf des Prädikats dürfen nur die Argumente *Ausgabezeichenkette* und *Code* ungebunden sein. Nach Beendigung des Editiervorgangs zeigt der Wert in *Code* an, wie das Editieren beendet wurde: Der Wert 0 besagt, daß die Funktionstaste <F10> gedrückt wurde, der Wert 1 zeigt an, daß das Editieren mit der <Esc>-Taste beendet wurde.

Grafik und Sound

Durch Grafik kann jedes Prolog-Programm aufgewertet werden. Beispielsweise könnte ein Expertensystem für technische Diagnosen bei Personalcomputern Grafik einsetzen, um das Bild einer Speicherplatine auszugeben und

darauf zu kennzeichnen, wo sich möglicherweise ein defekter Chip befindet. Die Grundidee beim Grafikkonzept in Prolog besteht darin, daß man einfache Prädikate zugrundelegt, um daraus komplexere Prädikate abzuleiten. Turbo Prolog stellt die wichtigsten Grafik-Prädikate zur Verfügung.

Daraus kann sich jemand, der beispielsweise Grafikspiele entwickelt, seinen eigenen Vorrat an Prädikaten definieren, z. B. für bewegte Objekte auf dem Bildschirm oder das Durchführen allgemeiner Operationen, die Bestandteil von vielen Spielen sind.

Die Prädikate dieses Kapitels (außer die für die Tonerzeugung) funktionieren nur mit CGA- oder EGA-Adaptern und -Monitoren. Wenn Sie mit einem Monochrom-Monitor arbeiten, können Sie gleich mit dem Abschnitt "Sound" am Ende des Kapitels fortfahren. Vielleicht lohnt es sich jedoch, das Kapitel wenigstens zu überfliegen, da einige der Grafik-Prädikate bei monochromen Grafikschirmen dazu dienen, die Auflösung zu beeinflussen.

Turbo Prolog unterstützt zwei Arten von Grafik. Bei der einen handelt es sich um das gewohnte Zeichnen von Linien und Punkten, die zweite ist die sogenannte "Turtle"-Grafik.

Linien und Punkte

Vier einfache Prädikate bilden den Befehlsvorrat zum Zeichnen von Linien und Punkten:

```
graphics(Modus,Palette,Hintergrund)

text

dot(Zeile,Spalte,Farbe)

line(Zeile1,Spalte1,Zeile2,Spalte2,Farbe)
```

Das Prädikat *graphics* schaltet in den Grafikmodus um, in dem dann die anderen Prädikate verwendet werden können. Dabei wird der Bildschirm gelöscht, der Cursor in der linken oberen Ecke plaziert, in die entsprechende Auflösung umgeschaltet, und die Vorder- und Hintergrundfarben werden gesetzt. Vor dem Aufruf des Prädikats *graphics* müssen alle seine drei Argumente gebunden sein. Mit dem Prädikat *text* können Sie hinterher wieder in den normalen Textmodus zurückkehren. Die Argumente des Prädikats *graphics* haben folgende Bedeutungen:

- Die Wahl des *Modus* steuert die Bildschirmauflösung (siehe Tabelle 15.3).
- Das Argument *Palette* wählt die Farbpalette für den Vordergrund aus (siehe Tabelle 15.4).
- Mit dem Argument *Hintergrund* wird die Hintergrundfarbe gewählt (siehe Tabelle 15.5).

Modus	Auflösung	Spalten	Zeilen	Farben	Adapter
1	mittel	320	200	4	CGA
2	hoch	640	200	(mono)	CGA
3	mittel	320	200	16	EGA
4	hoch	640	200	16	EGA
5	erweitert	640	350	13	EGA

Tabelle 15.3: Parameterwerte für das Argument Modus im Prädikat graphics

Palette Wert	Farbe 1	Farbe 2	Farbe 3
0	Grün	Rot	Gelb
1	Türkis	Lila	Weiß

Tabelle 15.4: Parameterwerte für das Argument Palette im Prädikat graphics (CGA)

Der *graphics*-Befehl leitet die Grafikverarbeitung nur ein; er selbst zeichnet nichts auf dem Bildschirm. Um eine Zeichnung auf dem Schirm anzufertigen, müssen Sie die Prädikate *dot* oder *line* einsetzen.

Das Prädikat *dot* kann sowohl Eingabe- als auch Ausgabe-Prädikat sein. Wenn das Argument *Farbe* beim Aufruf des Prädikats gebunden ist, wird es an die aktuelle Farbe der Position gebunden, die durch *Zeile* und *Spalte* festgelegt ist.

Jeder Punkt des Bildschirms hat zu einer bestimmten Zeit einen Farbwert, möglicherweise auch den gleichen wie der Hintergrund. Die Werte für *Zeile* und *Spalte* sind ganzzahlig und können von 0 bis 31999 reichen.

Das Prädikat *line* ist immer ein Ausgabe-Prädikat. Es zeichnet eine Linie zwischen *Zeile1,Spalte1* und *Zeile2,Spalte2*. Die Zeilen- und Spaltenwerte sind ebenfalls ganzzahlig und reichen von 0 bis 31999.

Hintergrund Wert	Farbe
0	Schwarz
1	Blau
2	Grün
3	Türkis
4	Rot
5	Lila
6	Braun
7	Weiß
8	Grau
9	Hellblau
10	Hellgrün
11	Helltürkis
12	Hellrot
13	Rosa
14	Gelb
15	Leuchtendweiß

Tabelle 15.5: Parameterwerte für das Argument Hintergrund im Prädikat graphics

Im Grafikmodus können außerdem die Prädikate *write* und *writef* für die Ausgabe von Text auf dem Bildschirm eingesetzt werden. Wenn Sie vorzugsweise mit Fenstern arbeiten, können Sie mit Hilfe des Prädikats *makewindow* ein oder mehrere Fenster definieren und Ihre Zeichnungen in diese Fenster setzen.

Turtle-Grafik

In Turbo Prolog steht uns noch eine zweite Möglichkeit für Grafikverarbeitung zur Verfügung: Turtle-Grafik. Dabei wird mit Hilfe einer stilisierten Schildkröte (engl. turtle) gezeichnet, die sich über den Bildschirm bewegt und dabei eine Spur hinterläßt. Die Schildkröte kann vorwärts, rückwärts, nach links oder nach rechts bewegt werden. Dabei kann der Zeichenstift abgehoben oder aufgesetzt und auf verschiedene Farben eingestellt werden. Sobald Sie in den Grafikmodus umgeschaltet haben, stehen Ihnen die Prädikate für Turtle-Grafik zusammen mit den anderen Prädikaten zum Zeichnen von Linien und Punkten zur Verfügung.

Um mit Turtle-Grafik zu arbeiten, müssen Sie genauso wie bei der anderen Grafikverarbeitung zunächst das Prädikat *graphics* aufrufen. Die Schildkröte befindet sich dann in der Mitte des Bildschirms und zeigt nach oben. Sie ist allerdings unsichtbar, und wir sehen nur die Spur, die sie beim Zeichnen hinterläßt. Die Turtle-Befehle kann man in zwei Gruppen aufteilen: in diejenigen zur Steuerung des Zeichenstifts und in die zur Steuerung der Bewegungen.

Steuerung des Zeichenstifts

Bevor wir mit der Zeichnung beginnen, müssen wir eine Stiftfarbe auswählen und den Stift senken. Dafür stehen uns drei Prädikate zur Verfügung:

pencolor(Farbe) legt die Farbe des Zeichenstifts fest. Diese kann beim CGA-Monitor Werte von 1 bis 3 annehmen. Zusammen mit der Palettenfarbe des Prädikats *graphics* (siehe Tabelle 15.4) wird dadurch die Stiftfarbe bestimmt.

pendown senkt den Stift zum Zeichnen.

penup hebt den Stift.

Steuerung der Bewegungen

Fünf Prädikate steuern die Bewegungen der Schildkröte:

forward(Schritt) bewegt die Schildkröte schrittweise vorwärts. Die Schrittgröße kann eine ganze Zahl zwischen 1 und 32000 sein. Die Schrittlänge hängt dabei noch vom gewählten Grafikmodus ab.

back(Schritt) bewegt die Schildkröte schrittweise rückwärts.

left(Winkel) dreht die Schildkröte gegen den Uhrzeigersinn um den angegebenen Winkel. Das Argument *Winkel* muß gebunden sein.

right(Winkel) dreht die Schildkröte im Uhrzeigersinn um den angegebenen Winkel. Das Argument *Winkel* muß gebunden sein.

penpos(Zeile,Spalte,Richtung) setzt die Schildkröte an die angegebene Position unter Berücksichtigung der vorgegebenen Richtung, wenn alle Argumente gebunden sind; wenn die Argumente frei sind, wird die aktuelle Schildkrötenposition zurückgegeben.

Sound

Turbo Prolog bietet zwei Prädikate zur Erzeugung von Tönen. Damit kann in einem Turbo Prolog-Programm eine Melodie gespielt werden.

beep gibt einen kurzen Piepston über den Lautsprecher

sound(Dauer,Frequenz) erzeugt eine Note mit der festgelegten Dauer und Frequenz. Dabei wird das Argument *Dauer* in Hundertstelsekunden angegeben. Bei *Frequenz* handelt es sich tatsächlich um die Frequenz in Hz. Dieser Wert reicht von 41 bis 17500 Hz.

Der Einsatz des Prädikats *beep* ist sinnvoll, wenn die Aufmerksamkeit des Anwenders, beispielsweise bei einem Fehler oder dem Ende der Verarbeitung, erregt werden soll.

Das Prädikat *sound* wird zur Erzeugung von Melodien verwendet. Seine beiden Argumente müssen gebundene ganzzahlige Werte sein. Die Frequenzwerte für eine Oktave können Sie der Tabelle 15.6 entnehmen.

Note	Frequenz
C (tief)	131
Cis	139
D	147
Dis	156
E	165
F	175
Fis	185
G	196
Gis	208
A	220
Ais	233
H	247
C (mittel)	262

Tabelle 15.6: Frequenzwerte für eine Oktave

Genau wie bei den Grafik-Prädikaten können Sie auch hier Ihre eigenen Prädikate unter Zugrundelegen dieser einfachen Prädikate bilden und als Bestandteil in Ihre persönliche Prädikaten-Bibliothek aufnehmen.

Übungen

1. Wie viele Zeilen und Spalten nimmt es in Anspruch, wenn Sie einen Rahmen um ein Fenster zeichnen? Unterscheidet sich diese Zahl für normale und inverse Rahmen?

2. Schreiben Sie ein kleines Programm, das die Meldung "Das ist ein Test." in einem Fenster von 8 Reihen Höhe und 20 Spalten Breite anzeigt. Die linke obere Ecke soll dabei in Zeile 10 und Spalte 30 liegen.

Kapitel 16

Dateiverarbeitung mit Turbo Prolog

Alle Programme, die wir bis jetzt geschrieben haben, waren in ihrer Größe durch den Rechnerspeicher beschränkt. Dabei befanden sich Turbo Prolog selbst, unser Quellprogramm und das compilierte Programm alle zusammen gleichzeitig im Speicher. Der dadurch verbrauchte Speicherplatz beschränkte die Größe unserer Programme und damit natürlich auch die Anzahl der Regeln und Fakten in unserer Datenbank.

Wesentlich größere Programme (Datenbanken) können erstellt werden, wenn Daten zwischenzeitlich auf der Platte gespeichert werden und das Programm Fakten von der Platte einliest. In diesem Kapitel wollen wir sehen, wie sich so etwas realisieren läßt.

Einführende Bemerkungen über Dateien

Dateien sind logische Speichereinheiten für zusammenhängende Daten. Computer-Dateien werden normalerweise als magnetische Aufzeichnungen auf einer Festplatte oder Diskette festgehalten. Jede Datei erhält einen Namen, der bis zu acht Zeichen lang sein kann, und eine durch einen Punkt abgetrennte Erweiterung von drei Zeichen. Auf einer Platte kann es mehrere Verzeichnisse geben, und jedes Verzeichnis kann mehrere Dateien enthalten. Der DOS-Befehl DIR zeigt eine Liste aller Dateien eines bestimmten Plattenverzeichnisses.

Turbo Prolog verfügt über mehrere Prädikate zum Lesen und Schreiben von Dateien. Auf den ersten Blick wirkt das Ganze vielleicht zunächst etwas schwierig, ist es aber im Prinzip nicht. Einige der Datei-Prädikate sind im Grunde nichts anderes als Schalter. Wenn sie aufgerufen werden, schalten sie die Eingabe von der Tastatur auf eine Datei um (oder irgendein anderes Gerät), oder sie lenken die Ausgabe vom Bildschirm in eine Datei um (oder auf ein anderes Gerät, z. B. auf einen Drucker).

Eine Datei zu lesen oder in eine Datei zu schreiben ist im Grunde nicht schwieriger als von der Tastatur zu lesen und auf den Bildschirm zu schreiben. Tatsächlich sind Tastatur, Bildschirm und Drucker für Turbo Prolog nichts anderes als Dateien.

Umlenken von Ein- und Ausgabe

Turbo Prolog stellt das Prädikat *readdevice* für das Umlenken der Eingabe und *writedevice* für das Umlenken der Ausgabe bereit. Beide besitzen ein einziges Argument, einen *symbolischen Dateinamen*.

Bestimmte symbolische Dateinamen sind für Standard-Systemgeräte reserviert (siehe Tabelle 16.1). Um beispielsweise die Ausgabe auf den Drucker umzulenken, dort eine Zeile zu drucken und dann die Ausgabe auf den Bildschirm zurückzulenken, geben Sie folgendes ein:

```
writedevice(printer),
write("Jetzt wollen wir drucken."),nl,
writedevice(screen).
```

Gerät	Symbolischer Dateiname	Ein-/Ausgabe
Tastatur	keyboard	Eingabe
Bildschirm	screen	Ausgabe
Drucker	printer	Ausgabe
Serieller Port	com1	Ein- oder Ausgabe

Tabelle 16.1: Standardnamen für Ein- und Ausgabe

Das Prädikat *writedevice* legt fest, wo die Ausgabe erfolgen soll, die von den Prädikaten *write* und *writef* erzeugt wird, und das Prädikat *readdevice* gibt an, von wo die Eingaben mit den Prädikaten zum Lesen (*readln, readchar* usw.) und ein paar speziellen Eingabe-Prädikaten, die wir noch besprechen werden, geholt werden sollen. Das Standard-Eingabegerät ist die Tastatur und das Standard-Ausgabegerät der Bildschirm.

Das angesprochene Gerät muß in jedem Fall startbereit sein. Wenn der Drukker angesprochen wird und er ist nicht angeschlossen, dann wird die Programmausführung an der Stelle des Ausgabe-Prädikats unterbrochen, und zwar so lange, bis der Drucker bereit ist. Genauso sollte der serielle Port bereit sein, wenn er angesprochen wird.

Die Prädikate *readdevice* und *writedevice* können auch für die Ein-/Ausgabeumlenkung in eine Datei verwendet werden. Die nächsten Abschnitte werden zeigen, wie das vor sich geht.

Öffnen und Schließen von Dateien

Die Verwendung von Plattendateien ist nur geringfügig komplizierter als das Arbeiten mit einem Drucker oder Ausgabe auf dem Bildschirm. Bevor eine Datei verwendet werden kann, muß sie *eröffnet* werden. Durch das Eröffnen einer Datei werden bestimmte Speicherbereiche für Datenübertragungen in eine Datei und von einer Datei bereitgestellt. Für Turbo Prolog sind auch Drucker, Tastatur und Bildschirm Dateien, die allerdings immer offen sind. Eine Plattendatei muß jedoch vor Aufruf des Prädikats *readdevice* explizit eröffnet werden.

Öffnen einer Datei

In Turbo Prolog gibt es mehrere Prädikate zum Öffnen einer Datei. Wir wollen in diesem Abschnitt zunächst einmal annehmen, daß die Datei zum Schreiben geöffnet werden soll, und deshalb brauchen wir das Prädikat *openwrite*. Die anderen Fälle werden im Abschnitt "Prädikate zum Öffnen von Dateien" aufgeführt.

Das Prädikat *openwrite* wird aufgerufen, bevor das Prädikat *writedevice* verwendet werden kann. Das allgemeine Format des Prädikats openwrite lautet:

```
openwrite(SymbolischerDateiname,Dateiname)
```

Das erste Argument ist der symbolische Dateiname, mit dem die Datei vom Programm aus angesprochen wird. Dieser symbolische Dateiname kann ein beliebiger Name sein (mit Ausnahme der in Anhang C aufgeführten), der die Syntaxregeln von Turbo Prolog erfüllt; d. h. er muß mit einem Kleinbuchstaben beginnen und kann Buchstaben, Ziffern und Unterstriche enthalten. Außerdem muß er im Abschnitt *domains* mit der Typbezeichnung *file* versehen werden.

Beispiele für symbolische Dateinamen sind:

```
xausgabe

namen

club_namen
```

Das zweite Argument, *Dateiname*, ist der Name für die Datei auf der Platte. Der Name ist ein Zeichenkettenwert mit höchstens acht Zeichen plus einer Erweiterung.

Beispiele für Dateinamen sind:

```
"test.dat"

"adressen.dat"
```

Wenn die Datei eröffnet ist, kann mit Hilfe des Prädikats *writedevice* die Ausgabe in die Datei umgelenkt werden, deren symbolischer Dateiname im Prädikat *openwrite* definiert wurde.

Schließen einer Datei

Wenn Ihre Arbeit mit einer Datei beendet ist, sollten Sie diese schließen. Beim Arbeiten mit Dateien befindet sich immer ein Teil der Datei im Rechnerspeicher und der andere Teil auf der Platte. Der Computer schreibt bzw. liest die Daten immer in Blöcken von einer festgelegten Größe auf die Platte bzw. von der Platte. Wenn Sie Programmdaten lesen und schreiben, sind diese jedoch von unterschiedlicher Länge. Während der Übertragung werden diese Daten kurzzeitig in speziellen Puffern des Rechnerspeichers zwischengespeichert. Wenn nun ein Programm endet, ohne daß die zugehörigen Dateien geschlossen worden sind, kann es vorkommen, daß Daten, die sich noch im Speicher befinden, verlorengehen und nicht auf die Platte geschrieben werden. Beim Schließen einer Datei werden alle zu dieser Datei zugehörigen temporären Pufferspeicher entleert und für andere Zwecke freigegeben.

Um sich vor dem Verlust von Daten zu schützen, sollte man also immer das Prädikat *closefile* aufrufen, um die Datei zu schließen, wenn sie nicht mehr gebraucht wird. Das allgemeine Format des Prädikats *closefile* lautet:

```
closefile(SymbolischerDateiname)
```

Nachdem die Datei geschlossen ist, sollten Sie die Ein- oder Ausgabe mit Hilfe des Prädikats *readdevice* oder *writedevice* wieder auf das Standardgerät zurücklenken.

Das Schließen einer Datei ist besonders wichtig, wenn Sie diese zum Schreiben verwendet haben. Sonst kann es passieren, daß einige Daten verlorengehen.

Beim Lesen einer Datei werden die Daten in ihr nicht verändert, so daß das Schließen hinterher keine so große Bedeutung hat. Man sollte aber trotzdem darauf achten, daß alle Dateien vor dem Programmende geschlossen werden, da es einen guten Programmierstil auszeichnet.

Ein Beispiel

An dem Beispiel aus Abb. 16.1 wollen wir betrachten, wie man die Prädikate praktisch anwendet. Als symbolischer Dateiname wurde in diesem Beispiel *xausgabe* gewählt. Der Name wird im Abschnitt *domains* mit der Typbezeichnung *file* versehen. Das Prädikat *openwrite* öffnet die Datei *test.dat* und ordnet ihr den symbolischen Dateinamen *xausgabe* zu. Als Folge davon stellt das Prädikat zwischen dem symbolischen Dateinamen und dem tatsächlichen Dateinamen eine logische Verbindung her und eröffnet die Datei, wobei der für die Datei notwendige Pufferbereich im Speicher bereitgestellt wird. Das Prädikat *writedevice* lenkt die Ausgabe auf diese symbolische Ausgabeeinheit um, und das Prädikat *write* führt den Schreibvorgang in diese Datei aus. Dann wird die Datei mit dem Prädikat *closefile* geschlossen. Die Ausgabe wird mit einem weiteren *writedevice*-Prädikat auf den Bildschirm zurückgelenkt.

```
domains
   file=xausgabe

predicates
   los

clauses
   los:-
         openwrite(xausgabe, "test.dat"),
         writedevice(xausgabe),
         write("Jetzt schreiben wir auf die
         Plattendatei."),nl,
         closefile(xausgabe),
         writedevice(screen).
```

Abb. 16.1: Ein einfaches Ausgabeprogramm

Zusammenfassend kann man den Ablauf folgendermaßen festlegen:

1. Die Datei wird eröffnet und der symbolische Dateiname mit dem tatsächlichen in Zusammenhang gebracht.
2. Mit den Prädikaten *readdevice* oder *writedevice* wird die Eingabe oder die Ausgabe auf die durch den symbolischen Namen gekennzeichnete Datei umgelenkt.
3. Die Ein- oder Ausgabeoperation wird durchgeführt.
4. Die Datei wird geschlossen.
5. Die Ein- oder Ausgabe wird auf die Tastatur oder den Bildschirm zurückgelenkt.

Prädikate zum Öffnen von Dateien

Im vorigen Abschnitt haben Sie das Prädikat *openwrite* kennengelernt, das eine Datei zum Schreiben öffnet. Bei der Verwendung dieses Prädikats wird jede Datei, die bereits den angegebenen Dateinamen besitzt, gelöscht, und das Schreiben erfolgt immer in eine neue Datei. Insgesamt gibt es vier Prädikate zum Öffnen von Dateien:

openread(SymbolischerDateiname, Dateiname) öffnet eine Datei zum Lesen. Wenn die Datei nicht existiert, scheitert das Prädikat.

openwrite(SymbolischerDateiname, Dateiname) öffnet eine Datei nur zum Schreiben. Eine bereits bestehende Datei mit gleichem Namen wird gelöscht.

openappend(SymbolischerDateiname, Dateiname) fügt neue Daten an die durch *Dateiname* angesprochene Datei an. Wenn es keine Datei mit diesem Namen gibt, wird die Ausführung beendet.

openmodify(SymbolischerDateiname, Dateiname) öffnet eine Direktzugriffs-Datei zum Lesen und Schreiben.

Wie Sie sehen, verfügen alle diese Prädikate über die gleiche Anzahl an Argumenten gleichen Typs. Alle dienen dem Öffnen einer Datei; dabei unterscheiden sie sich jedoch in der Festlegung, wie die weitere Verarbeitung der Datei erfolgen soll. Eine Datei, die mit dem Prädikat *openwrite* eröffnet wurde, kann z. B. nicht gelesen werden.

In einem Programm können mehrere Dateien gleichzeitig geöffnet werden. Obwohl die Prädikate *writedevice* und *readdevice* immer nur die Verwendung eines Eingabegerätes und eines Ausgabegerätes zur gleichen Zeit gestatten, so kann man doch beliebig zwischen den Geräten hin- und herschalten. Die Anzahl der Dateien, die zur gleichen Zeit offen sein dürfen, ist durch die Angaben *files* und *buffers* in der Systemdatei CONFIG.SYS beschränkt. Wenn Sie mehr Dateien benötigen und Schwierigkeiten auftreten, dann können Sie die Datei CONFIG.SYS dahingehend ändern, daß sie mehr Dateien und Puffer zuläßt, z. B.:

```
files = 20
buffers = 24
```

Entleeren der Puffer

Normalerweise werden die Plattenpuffer automatisch beim Schließen einer Datei entleert. Wenn eine Datei über lange Zeiträume hinweg für Eingaben geöffnet bleibt, sollte man dafür sorgen, daß sie zwischenzeitlich einmal geschlossen und dann wieder geöffnet wird. Dadurch kann dem Verlust von Daten bei Systemausfall vorgebeugt werden.

Eine mögliche Methode, die Speicherpuffer zu leeren, besteht also einfach darin, die Dateien bei normalem Programmende mit dem Prädikat *closefile* zu schließen. Eine andere Möglichkeit wäre, die eingegebenen Elemente zu zählen und abhängig von dieser Zahl die Datei von Zeit zu Zeit zu schließen und danach wieder zu öffnen.

Eine andere Möglichkeit bietet das Prädikat *flush*. Sein Format lautet:

```
flush(SymbolischerDateiname)
```

Es bewirkt, daß alle Daten aus den Puffern in die Datei übertragen werden, wobei diese geöffnet bleibt. Diese Vorgehensweise ist besonders beim Arbeiten mit seriellen Ports nützlich, wo das Zwischenschalten von Puffern Nachteile bringen kann.

Durch den Einsatz des Prädikats *flush* werden die Daten unmittelbar nach ihrer Eingabe an den Port geschickt. Auch sonst sollten Sie das Prädikat von Zeit zu Zeit aufrufen, wenn Sie mit längerfristig geöffneten Dateien arbeiten. Auf jeden Fall ist es eine zusätzliche Sicherung gegen Pufferüberlauf und damit einhergehenden Datenverlust.

Direkter Zugriff auf Dateien

Mit Hilfe des Standard-Prädikats *filepos* kann jede Position in der Datei direkt erreicht werden. Dann kann an dieser Stelle mit den Standard-Ein- und -Ausgabeprädikaten gelesen und geschrieben werden. Das allgemeine Format des Prädikats *filepos* lautet:

```
filepos(SymbolischerDateiname,Position,Modus)
```

Der Modus legt fest, von wo aus die Position errechnet werden soll:

0 vom Anfang der Datei

1 von der aktuellen Position

2 vom Ende der Datei

In der Abbildung 16.2 wird ein einfaches Beispiel für die Verwendung des Prädikats gezeigt.

```
domains
   file= eingabedaten

predicates
   los
   prüf_pos(char)
   repeat

clauses
   los:-
        write("Geben Sie den Dateinamen ein: "),
        readln(Dateiname),
        openread(eingabedaten, Dateiname),
        repeat,
        readdevice(keyboard),
        prüf_pos(X),
        X = '§',
        closefile(eingabedaten).

   prüf_pos(Daten):-
        nl,write("Position: "),
        readreal (Pos),
        readdevice(eingabedaten),
        filepos(eingabedaten,Pos,0),
        readchar(Daten),
        write(Daten).

   repeat.

   repeat:-
        repeat.
```

Abb. 16.2: Beispielprogramm für das Prädikat filepos

Der Ablauf des Programms ist folgender:

1. Die Datei wird durch das Prädikat zum Lesen geöffnet.

2. Die Position, an der gelesen werden soll, wird über die Tastatur eingegeben.

3. Die Eingabe wird durch das Prädikat *readdevice* auf die Datei umgelenkt.
4. Das Prädikat *filepos* steuert die Position an, an der gelesen werden soll.
5. Das Zeichen an der Position wird gelesen.
6. Die Eingabe wird für die weitere Verarbeitung auf die Tastatur zurückgelenkt.

Die Verarbeitung kann durch gleichzeitiges Drücken der Tasten <Alt>, <Ctrl> und <Break> abgebrochen werden. Beachten Sie, daß die Datei dabei nicht geschlossen wird. Da aber das reine Lesen einer Datei in dieser nicht zu Veränderungen führt, ist das Schließen in diesem Fall nicht so wichtig wie bei Dateien, in die geschrieben wurde.

Das Ende einer Datei kann während einer Leseoperation mit

```
eof(SymbolischerDateiname)
```

abgefragt werden. Dieser Test ist erfolgreich, wenn das Ende der Datei erreicht ist, andernfalls scheitert er.

Wenn Sie noch ausführlichere Informationen über Direktzugriffsdateien brauchen, sollten Sie in Ihrem Handbuch das Kapitel über dynamische Datenbanken lesen. Mit dieser Methode des direkten Zugriffs kann die Datenbank eines Programms als Plattendatei geführt werden, und damit ist ihre Größe nicht mehr auf den verfügbaren Speicherplatz im Rechner beschränkt.

Verwendung von Hilfsdateien

Manchmal kann es nützlich sein, für ein Programm eine Hilfsdatei bereitzustellen, die einem Anwender Informationen beim Einsatz des Programms geben kann. Hier bieten sich uns zwei Möglichkeiten.

Die erste Methode besteht in der Verwendung des Prädikats *file_str:*

```
file_str(Dateiname,Ausgabezeichenkette)
```

Dieses Prädikat liest bis zu 64K Zeichen aus der mit *Dateiname* angegebenen Datei in die *Ausgabezeichenkette*, und zwar bis das Endekennzeichen der Datei erreicht ist. Dann kann diese Zeichenkette mit dem Prädikat *display* auf dem Bildschirm angezeigt werden:

```
hilfe:-
    file_str('medizin.hlp',Hilfe),
    display(Hilfe).
```

Eine andere Möglichkeit bietet das Prädikat *editmsg* (siehe Kapitel 15), das eine Hilfsdatei logisch an die Funktionstaste <F1> bindet. Das Drücken von <F1> bringt dann diese Hilfsdatei auf den Bildschirm.

Dateioperationen in DOS

Die Dateioperationen auf DOS-Ebene umfassen sowohl das Löschen und Umbenennen von Dateien als auch Prüfoperationen auf Vorhandensein einer Datei. In Turbo Prolog können solche Dateioperationen entweder außerhalb eines Programms ausgeführt werden (z. B. beim Editieren) oder aber innerhalb eines Programms.

Externe DOS-Dateioperationen

Alle DOS-Kommandos (oder andere Anwendungsprogramme) sind jederzeit beim Editieren eines Programms verfügbar, ohne daß das Programm oder Turbo Prolog aus dem Speicher verlorengehen. Sie brauchen nur den Editor zu verlassen und das Files-Menü aufzurufen. Dort wählen Sie den letzten Punkt, *Operating system*, an. Sofort erscheint das DOS-Bereitschaftszeichen (z. B. C>).

Nun kann jedes beliebige DOS-Kommando eingegeben werden. (Vergewissern Sie sich aber, daß COMMAND.COM im Standard-Verzeichnis enthalten ist.) Nun können Sie das Inhaltsverzeichnis mit DIR ausgeben lassen, eine Datei mit REN umbenennen oder mit COPY kopieren. Sie können sogar ein anderes Anwendungsprogramm laden und ausführen, während Turbo Prolog und Ihr Programm weiterhin im Speicher verbleiben. Wenn Sie fertig sind, tippen Sie hinter dem Bereitschaftszeichen *exit* ein. Damit kehren Sie zu Turbo Prolog und Ihrem Programm zurück.

Ein Wort zur Vorsicht: Wenn Sie Turbo Prolog auf diesem Wege verlassen, wird im Speicher nichts gelöscht. Turbo Prolog bleibt im Speicher und wird lediglich inaktiv. Ihr Quellprogramm verbleibt ebenfalls im Speicher.

Wenn Sie nun ein anderes Programm ausführen wollen (DOS-Kommando, Anwendungsprogramm usw.), muß dieses sich mit dem verbleibenden Restspeicher begnügen. Vermutlich werden bei DOS-Kommandos keine Probleme auftreten, die ja entweder DOS-intern sind oder aber nur wenig Speicherplatz

benötigen. Wenn Sie aber versuchen, z. B. Lotus 1-2-3 oder ein anderes Anwendungsprogramm zu laden, kann es leicht passieren, daß der Speicherplatz knapp wird.

Anmerkung: Zur Vorsicht sollten Sie daher immer Ihr Programm sichern, bevor Sie Turbo Prolog verlassen. Der Speicherinhalt bleibt zwar erhalten, es kann aber immer vorkommen, daß die Ausführung eines Programms außerhalb von Turbo Prolog zu einem Systemausfall führt und Ihr Programm dann verloren ist.

Interne DOS-Dateioperationen

Beinahe alle DOS-Kommandos können innerhalb eines Programms aufgerufen werden. Zusätzlich stellt Turbo Prolog mehrere Standard-Prädikate für DOS-Funktionen zur Verfügung. Diese wollen wir als erstes nun betrachten.

Verwendung von Standard-Prädikaten zur Dateiverwaltung

Turbo Prolog besitzt vier Standard-Prädikate zur Dateiverwaltung:

existfile(Dateiname)	prüft das Vorhandensein der durch *Dateiname* spezifizierten Datei und ist erfolgreich, wenn diese im aktuellen Verzeichnis existiert (Typ *string*).
renamefile(Dateiname_alt,Dateiname_neu)	ändert den alten Dateinamen in den neuen (Typ *string*).
deletefile(Dateiname)	löscht die durch *Dateiname* angegebene Datei aus dem aktuellen Verzeichnis (Typ *string*).
disk(Pfad)	legt das zu verwendende Verzeichnis fest (Typ *string*).

Geben Sie das Programm aus Abb. 16.3 als Beispiel ein, und speichern Sie es unter dem Namen *test.pro*. Ändern Sie, falls nötig, die Laufwerksbezeichnung und den Pfadnamen entsprechend dem von Ihnen verwendeten Unterverzeichnis für das Prädikat *disk*, damit sie mit dem von Ihnen gewöhnlich verwendeten Laufwerk und Verzeichnis übereinstimmen. Dann können Sie das Programm ausführen.

```
predicates
   los

clauses
   los:-
        disk("C:"),
        existfile("test.pro"),
        renamefile("test.pro","dateineu.pro"),
        existfile("dateineu.pro"),
        deletefile("dateineu.pro"),
        write("Testprogramm für Dateioperationen.").
```

Abb. 16.3: Beipielprogramm mit Prädikaten zur Dateiverwaltung

Nach dem Programmstart (mit der Zielangabe *los*) wird die erste Bedingung der einzigen Regel aufgerufen. Diese richtet den Pfad ein und ist immer erfolgreich. Danach prüft das Prädikat *existfile* das Vorhandensein der Datei. Diese Überprüfung wird ebenfalls erfolgreich abgeschlossen, da die Datei unter dem Namen *test.pro* abgespeichert wurde. Die nächste Bedingung benennt die Datei um. Nun prüft Prolog das Vorhandensein der Datei mit dem neuen Namen. Danach wird die Datei mit dem neuen Namen gelöscht und die abschließende Meldung "Testprogramm für Dateioperationen" ausgegeben.

Wenn Sie das Programm erneut laufenlassen, wird es aufgrund der zweiten Bedingung scheitern, da nun keine Datei mehr mit dem Namen *test.pro* vorhanden ist.

Als abschließende Übung sollten Sie das Programm noch einmal unter dem Namen *test.pro* speichern und dann den Programmablauf mit *trace* verfolgen.

Aufruf von DOS-Kommandos im Programm

DOS-Kommandos können innerhalb eines Turbo Prolog-Programms über das Prädikat *system* aufgerufen werden. Fügen Sie beispielsweise das Prädikat *system* mit einer leeren Zeichenkette in ein Programm ein:

```
system(""),
```

Bei der Ausführung wird das Programm an dieser Stelle anhalten und das DOS-Bereitschaftszeichen ausgeben. Dann können Sie DOS-Kommandos wie RENAME oder COPY oder sogar ein Anwendungsprogramm aufrufen und ausführen. Mit *exit* kommen Sie danach zu Ihrem Programm zurück.

Das Prädikat kann auch selbst schon ein DOS-Kommando enthalten, wie zum Beispiel:

```
system("DIR A:"),
```

Dieses Kommando wird ein Verzeichnis aller Dateien auf der Platte in Laufwerk A ausgeben und dann mit der Ausführung des Programms fortfahren.

Verwendung von Kommandozeilen-Parameter

Es kann vorkommen, daß Sie die Parameter der Kommandozeile in einem Programm einsetzen möchten. Beispielsweise möchten Sie das Programm DIAGNOSE, das für eine externe Ausführung compiliert wurde, mit verschiedenen Datendateien verwenden. Um es für eine medizinische Diagnose einzusetzen, starten Sie es mit folgendem Kommando:

```
C>DIAGNOSE MEDIZIN
```

wobei MEDIZIN der Name der Datenbank-Datei ist, der natürlich auch wie hier in diesem Beispiel als Parameter in der Kommandozeile beim Aufruf eines Programms verwendet werden kann. Um diesen Parameter in ein Turbo Prolog-Programm zu übernehmen, schreibt man:

```
comline(Zeile)
```

Dabei wird *Zeile* an den Wert des Kommandozeilen-Parameters gebunden.

Übungen

1. Schreiben Sie ein kleines Programm, das Texte, die über die Tastatur eingegeben werden, an eine Datei anhängt.

2. Schreiben Sie ein Programm, um die Datei aus der ersten Übung wieder einzulesen und auf dem Bildschirm als sequentielle Zeilen auszugeben.

Kapitel 17

Programmentwicklung in Turbo Prolog

Bei der Programmentwicklung in Prolog gibt es verschiedene Techniken, um ein Programm effizient und zuverlässig zu machen. Turbo Prolog bietet außerdem noch zahlreiche Erleichterungen bei der Entwicklung von Programmen. Dieses Kapitel gibt eine Einführung in die grundlegenden Mechanismen bei der Programmierung in Turbo Prolog.

Programmierstil und Programmentwurf

Programmlistings sollten leicht lesbar sein. Bei der Prolog-Programmierung beginnt man mit einem kleinen Programmgerüst, das nach und nach erweitert wird. Da die Entwicklung eines Prolog-Programms sich über einen längeren Zeitraum erstrecken kann, sind Kommentare und eine geeignete Wahl von Variablennamen hier mindestens genauso wichtig wie bei Programmen in einer prozeduralen Sprache. Die Berücksichtigung der folgenden Anhaltspunkte bildet eine gute Ausgangsbasis für einen vorbildlichen Programmierstil und ein übersichtliches Layout:

- Wählen Sie die Namen für Prädikate, Objekte und Variablen so sinnvoll wie möglich. Definieren Sie die Namen für die Prädikat-Argumente des Abschnitts *predicates* im Abschnitt *domains*, anstatt die Prädikte über die verkürzte Form direkt mit Hilfe der Datentypen im Abschnitt *predicates* zu erwähnen. Verwenden Sie also im Abschnitt *predicates* beispielsweise

  ```
  ort(zimmernummer,beschreibung)
  ```

  anstelle von

  ```
  ort(integer,string)
  ```

- Geben Sie den Prädikaten Namen aus einem oder mehreren Wörtern, die vor allem die Beziehung zwischen den Programmobjekten ausdrücken und nicht nur die Funktion des Programms widerspiegeln. Die Bezeichnung *liesverb* ist beispielsweise ein geeigneterer Name für eine Syntax-Analyse als *analysieren* oder *liestoken*.

- Gewöhnen Sie sich eine einheitliche Terminologie an; z. B. können Sie Variablennamen in der Weise zusammensetzen, daß jedes Wort mit einem Großbuchstaben beginnt (z.B. *GanzeZahl*). Verbinden Sie mehrere Wörter in Relationsnamen durch einen Unterstrich (z. B. *hat_ein*).

- Bleiben Sie einem Stil treu, denn ein einheitlich aufgebautes Programm liest sich leichter.

- Richten Sie die Köpfe der Klauseln untereinander aus. Rücken Sie die Bedingungen ein, und beginnen Sie für jede eine neue Zeile. (Eine sehr kurze Liste mit Bedingungen kann unter Umständen in eine Zeile geschrieben werden, und zwar durch Kommas voneinander getrennt.)

- Fügen Sie Kommentare ein, die die Programmfunktionen erläutern.

- Geben Sie jedem Programm eine Überschrift mit Datum, Versionskennzeichnung, Verwendungszweck, Programmierername oder Quelle (z. B. Sybex-Buch) und benötigte Ablaufumgebung (z. B. EGA-Adapter und Prolog Version).

Programmiertricks und -techniken

Wenn Sie die folgenden Richtlinien befolgen, werden Ihre Programme effizient und funktionstüchtig sein:

- Gestalten Sie Ihr Programm so aussagekräftig wie möglich, um es überschaubar und leicht lesbar zu machen. (Vermeiden Sie unnötige Schnitte und Rekursionen.)

- Verwenden Sie in den Fällen, wo es möglich ist, lieber das Prädikat *not* anstelle eines roten Schnitts. Setzen Sie es jedoch mit Sorgfalt ein – es bewirkt nicht immer das, was Sie sich vorstellen. So kann es beispielsweise auch nicht als Teil eines Faktums oder in einem Regelkopf eingesetzt werden.

- Gebrauchen Sie die Diagnose-Optionen des Compilers und *check_determ*, um herauszufinden, wo grüne Schnitte sinnvoll zu plazieren sind.

- Die Programmentwicklung in Prolog ist ein zyklischer Prozeß. Die erste Programmversion ist ganz einfach und wird nach und nach TOP-DOWN schrittweise erweitert.

- Wenn eine Variable in einer Klausel nur einmal vorkommt, wird sie als anonym deklariert, z. B.:

```
los:-
annahme(_),,lösch_fakten.
```

- Wie in prozeduralen Programmen können Schalter für die Programmablauf-Steuerung gesetzt werden. Nehmen Sie den Namen des Schalters in die dynamische Datenbank auf, und setzen Sie ihn auf den Wahrheitswert True oder False. Verwenden Sie jedoch Schalter mit Zurückhaltung, da deren intensive Verwendung nicht unbedingt zu guten Programmen führt. Mit dem gleichen Trick können auch globale Variablen simuliert werden, die ein allgemein akzeptiertes Mittel zur strukturierten Programmablaufsteuerung und zur Datenübergabe zwischen verschiedenen Programmsystemteilen darstellen.

- Beschränken Sie die Anzahl der Prädikate in Ihrem Programm auf ein Minimum, und erhöhen Sie notfalls die Anzahl der Variablen. Das verbessert die Leistungsfähigkeit des Programms auf kleinen Rechnern, obwohl es das Programm unübersichtlicher macht.

- Ordnen Sie die Bedingungen im Regelrumpf so an, daß die Bedingungen mit den meisten gebundenen Variablen zuerst aufgeführt werden.

- Achten Sie darauf, daß ein Programm sinnvoll abgeschlossen wird, wenn kein Ziel und kein Teilziel erfolgreich ist.

- Vermeiden Sie prozedurale Routinen. Überlassen Sie die Hauptarbeit dem Unifikationsalgorithmus in Prolog.

- Rekursionen erhöhen die Anforderungen an den Stapelspeicher. Die Variablenwerte müssen in jeder Ebene für die Rückkehr aufbewahrt werden. Falls eine Rekursion unumgänglich ist, sollten Sie die Anzahl der Variablen in der rekursiven Klausel auf ein Minimum beschränken.

- Sorgen Sie dafür, daß Prädikat-Strukturen nicht zu lang werden. Wenden Sie hier ein Prinzip der strukturierten Programmierung an: Entwicklung vom Groben zum Feinen (top down) und Implementierung vom Feinen zum Groben (bottom up). Programmieren Sie modular.

- Versuchen Sie nicht, Standard-Prädikate (wie z. B. das Prädikat *not*) in Fakten oder Regelköpfen anzuwenden. In Prolog ist das mit diesen Prädikaten nicht möglich.

- Um die Ausführungszeit zu verkürzen, sollten Sie die Klauseln so anordnen, daß diese so schnell wie möglich scheitern.
- Schalten Sie Nicht-Determinismus aus, indem Sie explizite Bedingungen und Schnitte verwenden.

Verwendung von internen Zielen

Sie können eine Programmausführung dadurch vereinfachen, daß Sie ein *internes Ziel* definieren. Nach dem Programmstart wird dieses Ziel dann automatisch aufgerufen. Bei dieser Technik wird ein separater Abschnitt mit der Bezeichnung *goal* hinter dem Abschnitt *predicates* eingefügt. Die Definition eines internen Ziels muß mit einem Punkt abgeschlossen werden. Ein Beispiel sehen Sie in Abb. 17.1. Wenn Sie die Option *Run* wählen, wird das Programm automatisch ausgeführt und meldet nach Beendigung "Press the SPACE bar".

```
/* BEISPIEL MIT EINEM INTERNEN ZIEL */

domains
  name1, name2 = symbol

predicates
  mag(name1,name2)
  los

goal
  los.

clauses
  mag(hans,margret).
  mag(max,elke).

  los:-
      mag(max,_).
```

Abb. 17.1: Verwendung eines internen Ziels

Bei dieser Methode sucht Turbo Prolog immer nur nach einer Lösung für das interne Ziel. Wenn das Programm mehrere Lösungen suchen soll, kann man das Ziel in eine *fail-Schleife* einbetten, z. B.:

```
los:-
      annahme(X),
      fail.
los.
```

Compilierung für externen Gebrauch

Man kann ein Turbo Prolog-Programm so compilieren, daß es extern ausführbar wird, d. h. auf einem Computer ohne Turbo Prolog. Auf diese Weise könnte man Programmkopien an Leute verkaufen, die selbst Prolog nicht besitzen.

Um ein Programm dermaßen zu compilieren, müssen Sie folgendermaßen vorgehen:

1. Vergewissern Sie sich zunächst, daß bei Ihnen die Dateien PROLOG.LIB und INIT.OBJ1 im gleichen Verzeichnis stehen wie Turbo Prolog.

2. Entwickeln Sie Ihr Programm, und compilieren Sie es wie gewohnt, um sicher zu sein, daß es fehlerfrei läuft.

3. Fügen Sie ein internes Ziel ein, und definieren Sie dieses im Abschnitt *goal* (wie im vorigen Abschnitt dieses Kapitels beschrieben).

4. Machen Sie einen Probelauf mit dem internen Ziel, damit gewährleistet ist, daß es seine Funktion erfüllt.

5. Speichern Sie das Programm durch Auswahl des Punktes *Save* im Files-Menü.

6. Wählen Sie nun *Options* im Hauptmenü und dort

   ```
   EXE file (auto link)
   ```

7. Danach müssen Sie im Hauptmenü die Auswahl *Compile* treffen. Jetzt wird das Programm compiliert und mit einer Datei, die die Erweiterung .EXE führt, gelinkt. Dann werden Sie gefragt, ob das Programm ausgeführt werden soll. Wenn Sie mit "y" antworten, wird das Programm ausgeführt.

Wenn Sie nach der Programmausführung Turbo Prolog mit *Quit* verlassen, können Sie feststellen, daß sich im Inhaltsverzeichnis ein neues Programm befindet. Dieses trägt den Namen XXX.EXE, wobei XXX der Name Ihres Programms ist. Das Programm in dieser Datei kann jederzeit ohne Turbo Prolog ausgeführt werden, indem vom aktuellen Verzeichnis aus

```
XXX
```

eingetippt wird.

Da hier die externe Compilierung ein internes Ziel verwendet, wird das Programm nur nach einer Lösung für das Ziel suchen. Wenn Sie mehrere Lösungen brauchen, müssen Sie eine fail-Schleife einsetzen (siehe den vorigen Abschnitt "Verwendung von internen Zielen").

Die externe Compilierung ist genaugenommen ein Vorgang in zwei Schritten. Beide Schritte laufen automatisch ab, ohne daß der Benutzer eingreifen muß. Im ersten Schritt wird die eigentliche Compilierung ausgeführt und eine Datei mit dem Namen XXX.OBJ erzeugt. Ein integrierter Binder kombiniert diese Datei dann mit INIT.OBJ und Teilen von PROLOG.LIB und erzeugt letztendlich die Datei XXX.EXE.

Dazu wird nur das DOS-Binderprogramm (LINK.EXE) und kein zusätzlicher Binder speziell für Prolog benötigt. Um Ihnen ein Gefühl für Größenverhältnisse zu geben, bedenken Sie folgendes. Der Quellcode (Namenserweiterung .PRO) für das Expertensystem im nächsten Kapitel umfaßt ungefähr 7000 Bytes. Compiliert wird es zu einer Objekt-Datei (Namenserweiterung .OBJ) mit einem Umfang von ca. 11000 Bytes, und nach dem Binden (Namenserweiterung .EXE) sind es dann schon 56000 Bytes. Deshalb sollten Sie sich vor dem Compilieren vergewissern, daß genügend Speicherplatz zur Verfügung steht, um den Compilier-Vorgang erfolgreich abzuschließen.

Um Programme in Turbo Prolog mit Routinen in anderen Sprachen zu kombinieren, ruft man den Menüpunkt Options auf, mit dessen Unterpunkt *OBJ file* man dann die Datei zu einer Objekt-Datei compilieren kann. Dadurch wird der Compilier-Prozeß vor dem Binden beendet. Mit Hilfe eines externen Binders kann dann die Datei mit der Erweiterung .OBJ mit Objekt-Modulen, die in anderen Sprachen erstellt wurden, verbunden werden.

Programmablaufverfolgung

Der Ablauf eines Programms kann mit den beiden Compiler-Befehlen *trace* oder *shorttrace* überwacht werden. Der Befehl kann irgendwo vor dem Abschnitt *domains* eingefügt werden (siehe Abb. 17.2).

Ein Programm, das den Befehl *trace* oder *shorttrace* enthält und compiliert wurde, wird bei der Ausführung im Einzelschrittmodus arbeiten. Das heißt, nach jedem Schritt tritt so lange eine Unterbrechung ein, bis die Funktionstaste <F10> gedrückt wird. Dabei kann man anhand des Cursors und im Trace-Fenster den Weg durch das Programm bei der Suche nach Lösungen für das aktuelle Ziel mitverfolgen.

```
/* BEISPIEL FÜR DEN BEFEHL TRACE */
trace
domains
  name1,name2 = symbol

predicates
  mag(name1,name2)
  los

goal
  los.

clauses
  mag(hans,margret).
  mag(max,elke).

  los:-
      mag(max,_).
```

Abb. 17.2: Verwendung des Befehls trace

Der Befehl *shorttrace* ist dem Befehl *trace* vorzuziehen. Turbo Prolog setzt nämlich normalerweise einige Optimierungstechniken ein, um die Programmausführung effizienter zu machen. Diese bleiben beim Einsatz von *shorttrace* in Kraft. Wenn statt dessen *trace* verwendet wird, werden diese Optimierungstechniken ausgeschaltet, was die Ausführungsgeschwindigkeit verringert.

Eine andere Möglichkeit ist die Verwendung der Standard-Prädikate *trace(on)* und *trace(off)*, wenn man die Ablaufverfolgung an einer bestimmten Stelle im Programm ab- oder anschalten möchte. Wenn diese Schalter eingesetzt werden sollen, muß sich am Anfang des Programms entweder der *trace*- oder der *shorttrace*-Befehl befinden. Standardmäßig ist der Modus *trace(on)* voreingestellt.

Soll nur ein bestimmter Teil des Programms überwacht werden, so setzt man zunächst das Prädikat *trace(off)* irgendwo in die Nähe des Programmanfangs und ruft dann an der Stelle, wo die Programmablaufverfolgung einsetzen soll, *trace(on)* auf. Beendet wird die Ablaufverfolgung dann mit einem weiteren Prädikat *trace(off)*.

Es können auch nur speziell ausgewählte Prädikate überwacht werden. Diese müssen als Parameter beim Aufruf des Befehls *trace* oder *shorttrace* aufgeführt werden, zum Beispiel:

```
shorttrace symptom,diagnose
```

Test auf zusammengesetzte Fließmuster

Einige Prädikate können auf mehrere Arten verwendet werden. Beispielsweise kann *window_attr(Attribut)* Fensterattribute setzen, wenn *Attribut* gebunden ist. Wenn *Attribut* aber frei ist, liefert es die Attribute des aktuellen Fensters. Von solchen Prädikaten sagt man, daß sie ein *zusammengesetztes Fließmuster* haben. Auch einige selbstdefinierte Prädikate haben sicherlich solche Merkmale. In Kapitel 13 haben wir das Prädikat *mitglied* definiert (siehe Abb. 13.2), das ebenfalls ein zusammengesetztes Fließmuster aufwies, da es für verschiedene Typen von Listen brauchbar war.

Um in einem Programm solche zusammengesetzten Fließmuster herauszufinden, kann der Compiler-Befehl *check_cmpio* an den Anfang des Programms gesetzt werden.

Überlaufkontrolle

Bei sehr großen Programmen kann es beim Compilier-Vorgang dazu kommen, daß der Rechnerspeicher, der für das interne *Code*-Feld zur Verfügung steht, überläuft. Um das zu vermeiden, kann man den Befehl *Code* aufrufen, um den Platz für dieses Feld zu vergrößern. Standardmäßig ist dieser Bereich 16 KBytes groß. Der Parameter für den Code-Befehl gibt die Anzahl der Paragraphen an, die für diesen Bereich reserviert werden sollen. Ein Paragraph hat die Größe von 16 Bytes. Um also beispielsweise dem Feld 32 KBytes zu gewähren, müssen Sie schreiben:

```
Code = 2048
```

Warnungen und Unterbrechungen

Schon bei der Compilierung überprüft Turbo Prolog verschiedene Bedingungen für die Verwendung von Variablen. Sobald etwas Verdächtiges entdeckt wird, wird der Vorgang gestoppt und eine Warnung ausgegeben.

Die Verarbeitung kann durch Drücken der Funktionstaste <F10> fortgesetzt oder mit <Esc> abgebrochen werden. Turbo Prolog unterbricht z. B., wenn eine Variable in einer Klausel nur einmal auftaucht oder wenn eine Variable bei Rückkehr von einer Klausel nicht gebunden ist.

Wenn Sie beispielsweise das medizinische Diagnose-System aus Kapitel 12 so ändern, daß die höchstrangige Regel

```
los:-
    annahme(X),!,
    lösch_fakten.
```

lautet, so wird die Compilierung mit einer Warnmeldung unterbrochen. Dieses Problem kann man durch Verwendung einer anonymen Variablen umgehen:

```
los:-
    annahme(_),!,
    lösch_fakten.
```

Wenn Sie die Variable aber unbedingt mit ihrem Namen nennen wollen, so können Sie die Warnmeldung mit dem Compiler-Befehl *nowarnings* unterdrücken.

Während einer Programmausführung wird von Turbo Prolog vor jedem Prädikat die Tastatur daraufhin abgefragt, ob eine Unterbrechungsanforderung durch Drücken der Tasten <Ctrl> und <C> oder<Ctrl> und <Break> erfolgt ist, woraufhin die Ausführung sofort abgebrochen wird. Unter bestimmten Umständen kann es wünschenswert sein, diese Überprüfung abzuschalten, entweder um die Ablaufgeschwindigkeit zu erhöhen (da die Zeit für den Prüfvorgang wegfällt), oder wenn ein Programm auf keinen Fall unterbrochen werden darf (weil das vielleicht die Wissensbasis zerstören könnte).

Für diese Fälle kann die Tastatur mit dem Compiler-Befehl *nobreak* gesperrt werden, so daß der Anwender das Programm bei der Ausführung nicht unterbrechen kann. Wenn das compilierte Programm dann einmal gestartet ist, kann es nur noch durch Ausschalten des Systems gestoppt werden. Aus diesem Grund sollten Sie ein Programm, das diesen Befehl enthält, vor jeder Ausführung speichern.

Determinismus

Ein Prädikat, das aufgrund von Backtracking zu mehreren Lösungen führt, wird nicht-deterministisch genannt. Das Prädikat *repeat* ist beispielsweise von Natur aus nicht-deterministisch.

Unvorhergesehener Nicht-Determinismus von Prädikaten kann zu Fehlern beim Progammablauf führen. Nicht-deterministische Klauseln können durch den gezielten Einsatz eines Schnitts in deterministische umgewandelt werden. Der Compiler-Befehl *check_determ* leistet Unterstützung bei der Suche nach nicht-deterministischen Klauseln und somit beim richtigen Setzen der Schnitte.

Fügen Sie den Befehl irgendwo vor dem Abschnitt *domains* ein, und compilieren Sie Ihr Programm. Nicht-deterministische Klauseln werden daraufhin gekennzeichnet.

Bei den so markierten Klauseln wird das Programm seine Verarbeitung unterbrechen. Sie haben dann die Möglichkeit, die Funktionstaste <F10> zu drücken, um fortzufahren, oder mit <Esc> abzubrechen.

Sie können auch den Compiler-Befehl *diagnostics* verwenden, um nicht-deterministische Prädikate zu finden. Dieser Befehl gibt eine Tabelle für jedes Prädikat aus, in der Informationen über dessen Determinismus und andere Merkmale aufgeführt werden.

In der Abb. 17.3 wird die Ausgabe des *diagnostics*-Befehls als Bestandteil des Expertensystem-Beispiels aus Kapitel 19 gezeigt. Dabei bezieht sich der Begriff *flow pattern* darauf, wie die Argumente gebraucht werden.

Der Buchstabe *i* bedeutet, daß es sich um eine Eingabe (input) handelt und daß die Variable frei ist. Der Buchstabe *o* zeigt an, daß wir es mit einer Variablen für die Ausgabe (output) zu tun haben und daß diese gebunden ist.

```
Run     Compile     Edit     Options     Files     Setup     Quit

                          Diagnostics

Predicate Name  Dbase Determ Size  Dom1 -- flowpattern
--------------- ----- ------ ----- ------------------------
goal            NO    YES     89   --
xpositiv        YES   NO     116   symptom,integer -- i,i
xnegativ        YES   NO     116   symptom,integer -- i,i
menu            NO    YES    349   integer,integer,strlist,integer
                                     -- i,i,i,o
readkey         NO    YES     62   key -- o
readkey1        NO    YES    251   key,char,integer -- o,i,i
readkey2        NO    YES    395   key,integer -- o,i
maxlen          NO    YES    170   strlist,integer,integer -- i,i,o
listlen         NO    YES     80   strlist,integer -- i,o
writelist       NO    YES    117   integer,integer,strlist -- i,i,i
menu1           NO    YES    109   integer,integer,integer,integer
                                     -- i,i,i,o
menu2           NO    YES    345   integer,integer,integer,integer,key
                                     -- i,i,i,o,i

Use first letter of option  or  select with  ->  or  <-
```

Abb. 17.3: Verwendung des Prädikats diagnostics

Erstellen einer Prolog-Bibliothek

Sie werden voraussichtlich eigene Prädikate erstellen wollen, die in vielen Programmen eingesetzt werden sollen. Anstatt diese nun in jedem Programm neu einzutippen (und Tippfehler zu riskieren), sollten Sie diese Prädikate in einer Datei speichern. Beispielsweise können die Prädikate *mitglied, repeat, schreibliste, pause* und andere in einer separaten Datei mit den Abschnitten *domains, predicates* und *clauses* genauso wie jedes andere Programm abgespeichert werden.

Diese Datei kann später in jedes Programm mittels des Befehls *include*, der in jedem beliebigen Abschnitt stehen kann, eingelesen werden, und zwar wie folgt:

```
include"prolog.pro"
```

Ein Beispiel für die Anwendung von *include* finden Sie in Kapitel 19, wo das Programm *menu.pro*, das sich auf Ihrer Turbo Prolog-Diskette befindet, als Teil des Expertensystems genutzt wird. Dieses Menü-Programm wird über den Befehl *include* in das Expertensystem eingelesen.

Anmerkung: Es ist zu erwarten, daß ein Programm mit dem Einsatz von Bibliotheken an Effizienz verliert, aber die damit verbundenen Vorteile wiegen den Verlust wieder auf. Eine Alternative zum Befehl *include* stellt die folgende Technik dar: Beim Editieren eines Programms können Sie durch Drücken der Funktionstaste <F8> ein Hilfsfenster öffnen, in das Sie die Bibliotheksdatei hineinladen können. Kopieren Sie dann das gewünschte Prädikat in Ihr Programm hinein. Dann brauchen Sie nur noch die notwendigen Definitionen im Abschnitt *predicates* einzufügen.

Modulares Programmieren

Eine spezielle Eigenschaft von Turbo Prolog besteht darin, daß es modulare Programmierung unterstützt. Bei der Entwicklung eines umfangreichen Programms können damit z. B. Teilprobleme in verschiedenen Programm-Modulen gelöst werden. Das bringt einige Vorteile mit sich:

- Verschiedene Module eines Programms können gleichzeitig von mehreren Programmierern entwickelt werden.

- Neue Programme können auf bewährten Programmteilen, die bereits erfolgreich in anderen Programmen arbeiten, aufgebaut werden.

- Modulares Programmieren vereinfacht die Fehlersuche. Jedes einzelne Modul kann zunächst ausgetestet werden, bevor es in das endgültige Programm übernommen wird.

- Bei Programmänderungen brauchen lediglich die betreffenden Module verändert zu werden.

- Die Schnittstelle von Turbo Prolog-Programmen zu anderen Sprachen wird erleichtert. Die Module selbst können in verschiedenen Sprachen programmiert sein.

- Alle Prädikat- und Domain-Namen können entweder lokal oder global sein.

Anwendung von modularer Programmierung

Bei der modularen Programmierung müssen Sie eine Projekt-Datei definieren und dann die Programme unter Verwendung des Befehls *project* compilieren. Halten Sie dabei die folgende Vorgehensweise ein:

1. Definieren Sie ein Modul als das Hauptmodul. Das ist das einzige Modul, das ein Ziel enthalten kann.

2. Erzeugen Sie eine Projekt-Datei. Dazu wählen Sie den Punkt *Files* aus dem Hauptmenü aus und dort den Unterpunkt *Module list*. Dann erscheint die Eingabeaufforderung "Name of module list", und Sie müssen den Namen für die Modulliste eingeben. Den Suffix des Namens lassen Sie dabei weg. Nun befinden Sie sich im Editor-Fenster, das jetzt die Überschrift "Modules in Project" trägt. Tippen Sie die Namen aller Programmdateien ein, die in die Modulliste aufgenommen werden sollen. Lassen Sie auch diesmal die Namenserweiterungen weg, und schreiben Sie hinter jeden Namen ein Pluszeichen (+), zum Beispiel:

   ```
   test1+    test2+    test3+
   ```

   Wenn Sie damit fertig sind, drücken Sie die Funktionstaste <F10>, um das Menü zu verlassen. Die Projekt-Datei wird mit der Erweiterung .PRJ gespeichert.

3. Der Name des Projekts muß in jedem Modul mit dem Compiler-Befehl

   ```
   project"XXX"
   ```

   angegeben werden, wobei XXX der Name der Projekt-Datei ohne Suffix ist.

Verwendung von globalen Prädikaten und Domains

Standardmäßig sind alle Prädikate und Domains in einem Modul lokal. Auch wenn der gleiche Prädikat-Name in zwei Modulen vorkommt, so handelt es sich doch jedesmal um ein anderes Prädikat.

Wenn nun einige Prädikate für alle Module global verwendbar sein sollen, so müssen sie als globale Prädikate deklariert werden und in einem Abschnitt mit dem Titel *global domains* mit Typbezeichnungen versehen werden:

```
global domains
    elementeliste = teil*
    teil = symbol

global predicates
    repeat
    mitglied(element,elementeliste)
```

Sie sehen, daß für die Verwendung globaler Prädikate zwei neue Abschnitte erforderlich sind. Diese Abschnitte müssen in allen Modulen gleich sein und hinter den lokalen Abschnitten stehen, in denen die Prädikate verwendet werden.

Beispielsweise wird der Abschnitt *global predicates* hinter dem Abschnitt *predicates* und der Abschnitt *global domains* hinter dem Abschnitt *domains* eingefügt.

Eine andere Möglichkeit besteht darin, alle globalen Definitionen in eine eigene Datei zu schreiben und diese mit dem include-Befehl in jedes Modul einzubinden.

Aufbau eines Turbo Prolog-Programms

Wir haben gesehen, daß ein Programm in Turbo Prolog bis zu acht Abschnitte enthalten kann. Die meisten Programme verwenden davon nur ein paar, aber die vollständige Liste macht ein Programm flexibel für Erweiterungen: globale Prädikate für modulares Programmieren, dynamische Datenbanken, interne Ziele und Compiler-Befehle.

Auch wenn nicht alle Programme alle Abschnitte enthalten müssen, so ist es doch erforderlich, daß die vorhandenen eine bestimmte Reihenfolge einnehmen, damit das Programm erfolgreich compiliert werden kann:

1. Compiler-Befehle
2. domains
3. global domains
4. database
5. predicates
6. global predicates
7. goal
8. clauses

Übungen

1. Definieren Sie unter Verwendung der Standard-Prädikate für Datenbanken ein Prädikat, das einen Schalter darstellt.
2. Kann das Prädikat *not* eine Variable binden? Begründen Sie Ihre Aussage.

TEIL 2

Anwendungen

Kapitel 18

Einführung in Expertensysteme

Eine der Hauptanwendungen von Turbo Prolog besteht in der Entwicklung von Expertensystemen. In diesem und dem folgenden Kapitel werden Sie den vollständigen Ablauf für den Entwurf eines einfachen Expertensystems kennenlernen.

Das Expertensystem

Ein *Expertensystem* ist ein Computer-Programm, das mit Hilfe von Wissen und Schlußfolgerungs-Techniken Probleme lösen kann, die normalerweise menschlichen Sachverstand erfordern. Ein Expertensystem besteht aus einer großen Anzahl von Fakten und dazugehörigen Regeln, deren Auswertung zu Lösungen führen kann. Eine solche Sammlung von Informationen wird auch *Wissensbasis* genannt.

Im Gegensatz zu herkömmlichen Computer-Programmen können mit einem Expertensystem auch Probleme bearbeitet werden, die unstrukturiert sind und für deren Lösung sich keine formale Prozedur anbietet. Das Expertensystem erstellt unter Bezugnahme auf die Wissensbasis eine eigene Prozedur, die der Problemdefinition gerecht wird. Es eignet sich daher besonders für Fälle, in denen formale Schlußfolgerungen zum Ziel führen.

Aufgaben des Expertensystems

Die besten Anwendungsmöglichkeiten für Expertensysteme liegen auf dem Gebiet von Diagnosen, Vorhersagungen, Analysen, Steuerungen, Planungen und Lernprogrammen.

In den meisten Fällen unterstützen Expertensysteme Fachleute, die schnelle und objektive Entscheidungen fällen müssen, bei der Durchsicht großer Datenbanken mit Fachinformationen. Eine zunehmende Tendenz für den Einsatz von Expertensystemen wird jedoch auch auf den Gebieten verzeichnet, bei denen menschliches Fachwissen zu kostspielig oder überhaupt nicht ausreichend verfügbar ist.

Im Vergleich zum menschlichen Gehirn speichert ein Computer vergleichsweise kleine Mengen an Wissen. In einem Expertensystem wird im allgemeinen Wissen nur über einen einzigen Themenbereich gespeichert, der auch *Domain* genannt wird. Der Computer kann deshalb nur Probleme analysieren und lösen, die diesen Bereich betreffen. Da die Mikrocomputer zur Zeit noch über sehr begrenzte Speicherkapazitäten verfügen, ist es für Anwendungen in Turbo Prolog sinnvoll, diesen Bereich so eng wie möglich abzustecken. Realisierbare Anwendungen für Expertensysteme sind z. B.:

- Ein umfassendes medizinisches Diagnose-System würde sogar die Kapazitäten eines Großrechners sprengen, aber ein System zur Analyse von Lungenkrankheiten würde beispielsweise in einen Minicomputer passen.

- Ein Diagnose-System für Waschmaschinen kann ebenfalls sinnvoll sein, wenn sich dieses auf einen bestimmten Hersteller oder ein bestimmtes Modell beschränkt.

Die folgenden Anwendungen für Expertensysteme können mit Turbo Prolog auf einem Mikrocomputer implementiert werden:

- Diagnose-Programm für die Reparatur eines Personal-Computers. Durch Auswechseln der Datenbank kann dasselbe System zur Diagnose bei Problemen mit Waschmaschinen, Autos, Videorecordern oder Kameras eingesetzt werden.

- Entwurf einer Broschüre, die mit Hilfe eines Desktop-Druckprogramms erstellt werden soll.

- Planung eines Telekommunikations-Systems für eine Firma.

- Anzeige der Temperatur und Luftfeuchtigkeit der verschiedenen Räume eines Hauses und optimale Steuerung der Zentralheizungsanlage.

- Planung einer Investitionstaktik.

- Erstellen von Diagnosen in medizinischen Teilbereichen.

- Unterstützung von Maschinenbedienung durch Bedienerführung und Vorschläge bei Ausnahmesituationen.

- Kreditsachbearbeitung bei Konsumentenkrediten.

- Beratung bei der Baufinanzierung.

- Beratung bei der Altersversorgung.
- Beratung bei der Wertpapieranlage.
- Unterstützung beim Datenbank-Design.
- Analyse von Kommunikationsflüssen für Bürokommunikationssysteme.
- Schätzung des Programmieraufwandes für DV-Anwendungen.

Der Wissensverarbeitungs-Fachmann

Das größte Problem beim Entwurf eines Expertensystems besteht darin, das Wissen eines Experten zu abstrahieren und in eine objektive Form für die Aufnahme in der Datenbank zu bringen. Der Experte geht an ein Problem immer auch mit teilweise subjektiven Vorstellungen heran. Im Gegensatz dazu kann ein Computer nur mit objektiven Darstellungen arbeiten. Erst wenn formale Regeln und Fakten existieren, kann ein Expertensystem errichtet werden.

Expertensysteme werden von Wissensverarbeitungs-Fachleuten entworfen. Diese müssen im Kontakt mit dem Experten festlegen, wie dessen Gedankengänge in objektive Formen zu lenken sind. Dabei gibt es keine allgemeinen Theorien oder Regeln, die bei dem Entwurf eines Expertensystems zugrundegelegt werden können. Das Problem, das der Experte zu lösen versucht, ist oft noch unstrukturiert, und es liegen häufig keine bekannten formalen Lösungswege vor.

Wissensdarstellung

Ein Expertensystem speichert Daten über Dinge des täglichen Lebens. Bei der Wissensdarstellung werden diese alltäglichen Dinge zu *Objekten*. Jedes Objekt besitzt ein oder mehrere *Attribute* oder Eigenschaften, und diese Attribute haben *Werte*; z. B.:

Objekt	**Attribut**	**Wert**
Auto	Farbe	Blau
Tisch	Form	Rund

Ein solches Objekt ist, wie wir bereits erwähnt haben, nicht das gleiche wie ein Objekt in Prolog.

Fakten in der Datenbank können diese drei Begriffe – Objekt, Attribut, Wert – zusammenfassend darstellen, wie z. B. als

```
symptom(rolf,fieber(hoch)).
```

oder in manchen Fällen auch nur zwei der Begriffe, nämlich Attribut und Wert, als

```
symptom(fieber,hoch).
```

Dabei ist *rolf* das Objekt, *fieber* das Attribut und *hoch* der Wert.

In der gleichen Weise können Regeln Beziehungen zwischen den drei Begriffen Objekt, Attribut und Wert darstellen, oder auch wieder nur zwischen Attribut und Wert.

Typen von Expertensystemen

Zwei Haupttypen von Expertensystemen können mit Prolog erstellt werden. Dabei handelt es sich einmal um das sogenannte *Produktionssystem* und zum anderen um das *rahmenbasierende System*. Den Schwerpunkt dieses Kapitels bildet das Produktionssystem, aber zum Abschluß wird auch ein kurze Einführung in rahmenbasierende Systeme gegeben.

Produktionssysteme

Ein Produktionssystem ist ein Expertensystem, bei dem das Wissen in Regeln und Fakten gespeichert wird. Jede Regel ist eine *Produktionsregel*. Die Regeln drücken Beziehungen zwischen den Fakten aus.

Da Sie bereits wissen, wie Regeln und Fakten in Prolog gehandhabt werden, können Sie sich vielleicht ein Bild davon machen, wie einfach es sein wird, ein Produktionssystem in Prolog zu implementieren.

Ein Produktionssystem besteht aus zwei wichtigen Komponenten: aus der Wissensbasis und dem Schlußfolgerungs-Automatismus (Abb. 18.1).

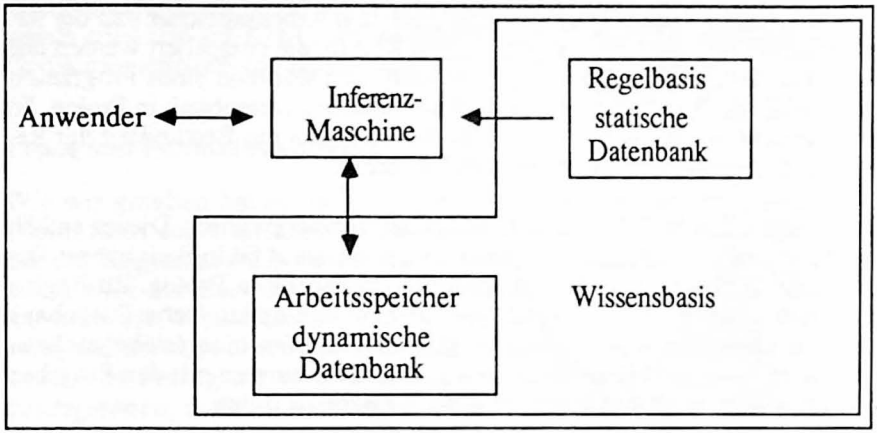


Abb. 18.1: Das Produktionssystem

Die Wissensbasis

Eine *Wissensbasis* sind die Daten oder das Wissen, die die Grundlage für das Fällen von Entscheidungen bilden. Die Wissensbasis enthält die Regeln und Fakten über das Gebiet. Die Darstellung in einer solchen Regelbasis ermöglicht es einem Expertensystem, ein Problem auf ähnliche Weise wie ein menschlicher Experte zu lösen:

`IF` (Wenn) der Patient einen krächzenden, keuchenden Husten hat

`AND` (und) niest

`AND` (und) Schnupfen hat

`AND` (und) an Appetitlosigkeit leidet,

`THEN` (dann) hat er vermutlich Keuchhusten.

Dieser Sachverhalt kann in Prolog folgendermaßen dargestellt werden:

```
annahme(keuchhusten):-
    symptom(husten),
    symptom(niesen),
    symptom(schnupfen),
    symptom(appetit).
```

Die Wissensbasis besteht aus zwei Teilen: aus dem Arbeitsspeicher und der Regelbasis. Die *Regelbasis* enthält Fakten und Regeln, die compiliert werden und einen Teil des Problems bilden. Sie verändern sich während eines Programmlaufs nicht. Die Regelbasis entspricht einer statischen Datenbank in Prolog. So ist beispielsweise die vorige Regel über Keuchhusten ein Bestandteil der Regelbasis beziehungsweise der statischen Datenbank.

Den zweiten Teil der Wissensbasis bildet der *Arbeitsspeicher*. Dieser enthält die Fakten, die nur für einen bestimmten Programmlauf Gültigkeit haben. Der Arbeitsspeicher entspricht der dynamischen Datenbank in Prolog. Zu Beginn einer Befragung ist dieser Arbeitsspeicher bzw. die dynamische Datenbank leer. Im weiteren Verlauf verwendet die Inferenz-Maschine (siehe nächster Abschnitt) Fakten und Regeln aus der Regelbasis zusammen mit den Eingaben des Anwenders, um den Arbeitsspeicher mit Fakten zu füllen.

Der Schlußfolgerungs-Automat

Die in Prolog realisierte Schlußfolgerungs-Maschine, auch Inferenz- Maschine genannt, hat die beiden Funktionen *Schlußfolgerung* und *Steuerung*. *Inferenz* ist ein formaler Schlußfolgerungs-Mechanismus. Er schließt einen Unifikationsalgorithmus mit ein und verwendet bekannte Fakten, um neue Fakten mit Hilfe von Regeln daraus abzuleiten. Ein solcher Schlußfolgerungs-Automatismus arbeitet in einem *Modus ponens*. Dies ist ein Grundelement der formalen Logik. Vereinfacht ausgedrückt, bedeutet *Modus ponens* für eine Regel mit dem Wortlaut

```
IF A THEN B
```

daß, wenn A wahr ist, B auch wahr ist.

Die Steuerungsfunktion legt fest, in welcher Reihenfolge die Regeln überprüft werden und was passieren soll, wenn eine Regel erfolgreich ist oder scheitert. Die Inferenz-Maschine nimmt die Fakten, von denen sie weiß, daß sie wahr sind, aus der Regelbasis (statische Datenbank) und aus dem Arbeitsspeicher (dynamische Datenbank) und verwendet diese, um die Regeln in der Datenbank mit Hilfe des Unifikationsprozesses zu überprüfen. Wenn eine Regel Erfolg hat, sagt man, sie wird *entlassen*, und der Regelkopf wird im Arbeitsspeicher aufgenommen.

In einigen Fällen kann es vorkommen, daß mehr als eine Regel mit den bekannten Fakten unifiziert. Dann muß die Steuerungsfunktion entscheiden, welche Regel entlassen wird. Dieser Fall wird auch *Konfliktlösung* genannt. Die

Steuerungsfunktion ist jedoch eine zyklische Funktion. Sobald eine Regel entlassen wird und ihr Kopf zum Arbeitsspeicher hinzugefügt wird, beginnt der Unifikationsprozeß von neuem, und der Zyklus wird wiederholt.

Prolog und Produktionssysteme

Wie wir gesehen haben, ist Prolog im Kern ein Schlußfolgerungs-Automat. Dieser führt den Unifikationsalgorithmus durch, legt die Reihenfolge fest, in der die Regeln geprüft werden sollen, und handhabt Konfliktlösungen. Dadurch wird der Entwurf eines Expertensystems verhältnismäßig einfach, da nur noch die Wissensbasis erstellt zu werden braucht. Natürlich bedeutet das auch, daß der Anwender weniger Steuerungsmöglichkeiten hat. Beispielsweise werden in Prolog die Regeln nach dem Prinzip der *Rückwärtsverkettung* durchgesehen; d. h. mit dem Ziel wird begonnen und dann rückwärts gesucht. Prolog geht dabei auch nach dem *Tiefensuchverfahren* vor. Alle Regeln, die für ein bestimmtes Ziel in Frage kommen, werden so tief wie möglich nach einer Lösung abgefragt, bevor Prolog durch Backtracking zurückgeht und ein anderes Ziel versucht.

Im praktischen Systementwurf bringt der Anwender lediglich ein paar zusätzliche Steuerungsregeln mit ein, die nicht Bestandteil der eigentlichen Wissensbasis sind. Rekursionen, Schnitte und das Prädikat *fail* sind alles, was Prolog an Steuerungsmöglichkeiten zu bieten hat. Beim Entwurf eines Expertensystems werden Sie sicher bestimmte Prädikate und Regeln zu Steuerungszwekken entwickeln. Dann enthält die statische Datenbank tatsächlich zwei Typen von Regeln, die der Wissensbasis und die Steuerungsregeln.

Wenn Sie sich einmal das Expertensystem-Beispiel GENI anschauen, das Sie zusammen mit Ihrem Turbo Prolog-Programm erhalten haben, werden Sie die Prädikate gruppiert und klassifiziert finden. Bei den meisten Prädikaten handelt es sich um Prädikate zu Steuerungszwecken (Befehle, Inferenz-Regeln usw.).

Die Anzahl der Prädikate für die Wissensbasis ist sehr begrenzt, aber dafür wird jedes mehrfach verwendet. Zusammenfassend kann man sagen, daß ein großes Expertensystem im allgemeinen wesentlich mehr Regeln für die Wissensbasis als für Steuerungszwecke umfaßt.

Zusätzliche Möglichkeiten

Besonders komplexe Expertensysteme können noch zusätzliche Komponenten enthalten, wie z. B. eine natürlich-sprachliche Schnittstelle oder ein Erläuterungssystem.

Durch eine natürlich-sprachliche Schnittstelle wird die Kommunikation des Anwenders mit dem System wesentlich verbessert. Bei den meisten Abfragesystemen kann der Anwender mit ja oder nein antworten, eine Auswahl in einem Menü mit mehreren Wahlmöglichkeiten treffen oder den Wert für ein Attribut eingeben (z. B. das Alter des Patienten). Eine natürlich-sprachliche Schnittstelle (siehe Kapitel 1) ermöglicht dem Anwender, während des Abfrageprozesses mit dem Expertensystem mittels einer Untermenge seiner Mutter- oder Fachsprache einen Dialog zu führen.

Ein *Erläuterungssystem* erlaubt dem Anwender auf eine Frage mit dem Wort *warum* zu antworten. Das System gibt dann eine Information über das Ziel, das gerade bewiesen werden soll, und hilft dadurch dem Anwender, den Überlegungsprozeß des Systems zu verstehen.

Rahmenbasierende Systeme

Ein anderer Typ eines Expertensystems ist das rahmenbasierende System. In diesem System wird das Objekt durch einen *Rahmen* dargestellt. Der Rahmen enthält (analog zu Schubladen einer Kommode) ein oder mehrere *Füll-Slots*, wobei jedes Slot (Schublade) ein Attribut vertritt. Attributwerte werden in diesen Schubladen gespeichert. Eine Schublade kann auch z. B. Grenzwerte für andere Schubladen enthalten.

Der Rahmen repräsentiert eine sehr komplexe und leistungsfähige Form einer Wissensdarstellung. Rahmen können in hierarchische Beziehungen (Analogie Anbauwand) gesetzt werden, die ein *Klassifikationssystem* ausdrücken. Slots können sogar Prozeduren enthalten, die dazu aktiviert werden können, für ein Attribut einen numerischen Wert zu berechnen. Sie können ebenso Regeln enthalten, die einen Schlußfolgerungsprozeß aktivieren, der einen Attributwert ergibt.

Stellen Sie sich einmal ein *Produktionssystem* vor, das mit der Zeit immer umfangreicher wird. Es enthält möglicherweise mehrere hundert Regeln, deren Beziehungen untereinander nur noch schwer zu überblicken sind. Wenn neue Regeln hinzugefügt werden, widersprechen diese vielleicht den bereits vorhandenen. In einem rahmenbasierenden System kann die Reihenfolge und die Beziehung der Elemente zueinander immer im Auge behalten werden, da die Reihenfolge einen Teil der Wissensbasis darstellt und im Systementwurf verankert wird.

Rahmenbasierende Systeme eignen sich besonders gut für die Anwendung in biologischen Klassifizierungssystemen und ähnlichen Systemtypen, in denen eine statische hierarchische Ordnung einen Teil des Wissens ausmacht.

Prolog gestattet die Verwendung einer rahmenbasierenden Darstellung, aber um die rahmenbasierenden Schlußfolgerungen zu unterstützen, muß man seine eigenen Prädikate schreiben.

Anmerkung: Rahmenbasierende Systeme unterstellen, daß die hierarchische Beziehung der Objekte untereinander relativ statisch ist. Wenn diese Ordnung dynamisch ist, wird die Verwendung eines rahmenbasierenden Systems ziemlich schwierig.

Übungen

1. Finden Sie im medizinischen Diagnose-System, das in Kapitel 10 entwickelt wurde, den Arbeitsspeicher und die Regelbasis heraus.

2. Bestimmen Sie in dem System aus Kapitel 10 die Prädikate zu Inferenz- und Steuerungszwecken und die der Wissensbasis. Wie viele Regeln enthalten diese beiden Gruppen jeweils?

3. Nennen Sie die Vor- und Nachteile bei der Verwendung von Turbo Prolog als Inferenz-Maschine für ein Expertensystem.

Kapitel 19

Aufbau eines Expertensystems

In diesem Kapitel werden Sie den Entwurfsablauf eines einfachen Expertensystems nachvollziehen können. Mit ähnlichen Techniken können Sie dann selbst nahezu jedes diagnostische Expertensystem aufbauen.

Festlegen des Fachgebiets und Formulierung der Zielaussage

Die erste Aufgabe besteht darin, objektiv festzulegen, was eigentlich erreicht werden soll. Der Fachbereich und auch die Objekte und Attribute in diesem Bereich müssen definiert werden. Dabei sollte der Bereich so eng umgrenzt sein, daß mit Hilfe von Turbo Prolog alle Regeln und Fakten, die gebraucht werden, im Rechner Platz finden.

Wenn der zu bearbeitende Problembereich mehr Fakten und Regeln erfordert, als mit Turbo Prolog verwaltet werden können, kann man zunächst versuchen, einen Teilbereich herauszunehmen und mit diesem unter Turbo Prolog ein Mustersystem aufzubauen. Ein solcher Prototyp ist oft in der Lage, dem Entwickler Einblicke in die Problematik zu verschaffen und als Zwischenschritt bei der Lösung größerer Probleme wertvolle Dienste zu leisten.

Wir wollen als Beispiel noch einmal das medizinische Diagnose-System für Kinderkrankheiten, das wir bereits in früheren Kapiteln angeführt haben, betrachten und weiterentwickeln. Dieses System sollte den Anwender über Krankheitssymptome befragen und dann zu einem von acht möglichen Ergebnissen kommen: Der Patient hat Masern, Röteln, Mumps, Windpocken, Gehirnhautentzündung, Keuchhusten, Grippe oder eine einfache Erkältung.

Anmerkung: Wir weisen darauf hin, daß dieses System nur der Veranschaulichung grundlegender Entwurfstechniken für Expertensysteme dienen soll. Es sollte nicht für den professionellen Einsatz in der Medizin verwendet werden und wurde auch nicht von einem Arzt oder einer medizinischen Organisation geprüft.

Man kann den Schlußfolgerungsprozeß eines Expertensystems als die Suche durch einen *Problemraum* betrachten. Dabei repräsentiert der Raum das Gebiet der Wissensbasis (siehe Abb. 19.1). Auf dem Weg zum eigentlichen Ziel liegen die Subziele. Das in Prolog vorliegende heuristische Verfahren beginnt mit einem Ziel (eine spezielle Krankheit) und versucht, dieses zu beweisen, indem es die Subziele beweist. Wenn der Beweis scheitert, geht Prolog zum nächsten Ziel über und versucht, dieses zu beweisen. Die Hauptaufgabe des Wissensverarbeitungs-Fachmanns besteht darin, diese Subziele zu definieren.

Der Problemraum besteht aus *Knoten* und *Verbindungen*. Dabei steht jeder Knoten sowohl für ein Symptom als auch für ein Subziel. In unserem Diagnose-System kann ein Knoten beispielsweise das Symptom *Fieber* repräsentieren. Dieser Knoten wird in dem Moment erreicht, wenn dem Anwender die Frage gestellt wird, ob der Patient Fieber hat. Außerdem stellt dieser Knoten ein Subziel für alle Krankheiten dar, die das Symptom Fieber aufweisen. Die Verbindungen sind die Operationen. Durch Fragenstellen gelangt das System von einem Knoten zum anderen.

Der Entwurf eines Expertensystems ist immer ein zyklischer Vorgang. Wenn Fachleute ein Problem lösen sollen, versuchen sie dieses zunächst in kleinere Teilprobleme aufzugliedern. Wenn die Stereo-Anlage nicht funktioniert, wird der Fachmann zuerst prüfen, ob der Fehler an den Lautsprechern, an den Kabeln, am Hauptgerät oder am Eingabeträger (Band, Platte usw.) liegt. Beim Entwurf von Expertensystemen wird in ähnlicher Weise verfahren. Das Unterteilen des Problemraums führt zu Subzielen. Dabei bringt der erste Aufteilungsversuch vielleicht noch nicht die geeignetesten Subziele mit sich, aber er stellt auf jeden Fall einen Anfang für den Entwurfsprozeß dar.

Beschaffung der Fakten

Sobald das Ziel des Systems feststeht, besteht die nächste Aufgabe darin, eine Liste der Fakten anzulegen, die erforderlich sind, um jede mögliche Bedingung zu erfüllen. Eine Liste mit Fakten, die für die verschiedenen Entscheidungen im medizinischen Diagnose-System gebraucht werden, ist in Abb. 19.2 aufgeführt. Sie werden feststellen, daß in diesem wie in den meisten Diagnose-Systemen die höchstrangigen Regeln von der Form

```
annahme(Z):-
    symptom(X1),
    symptom(X2),
    .   .   .
    symptom(Xn).
```

sind.

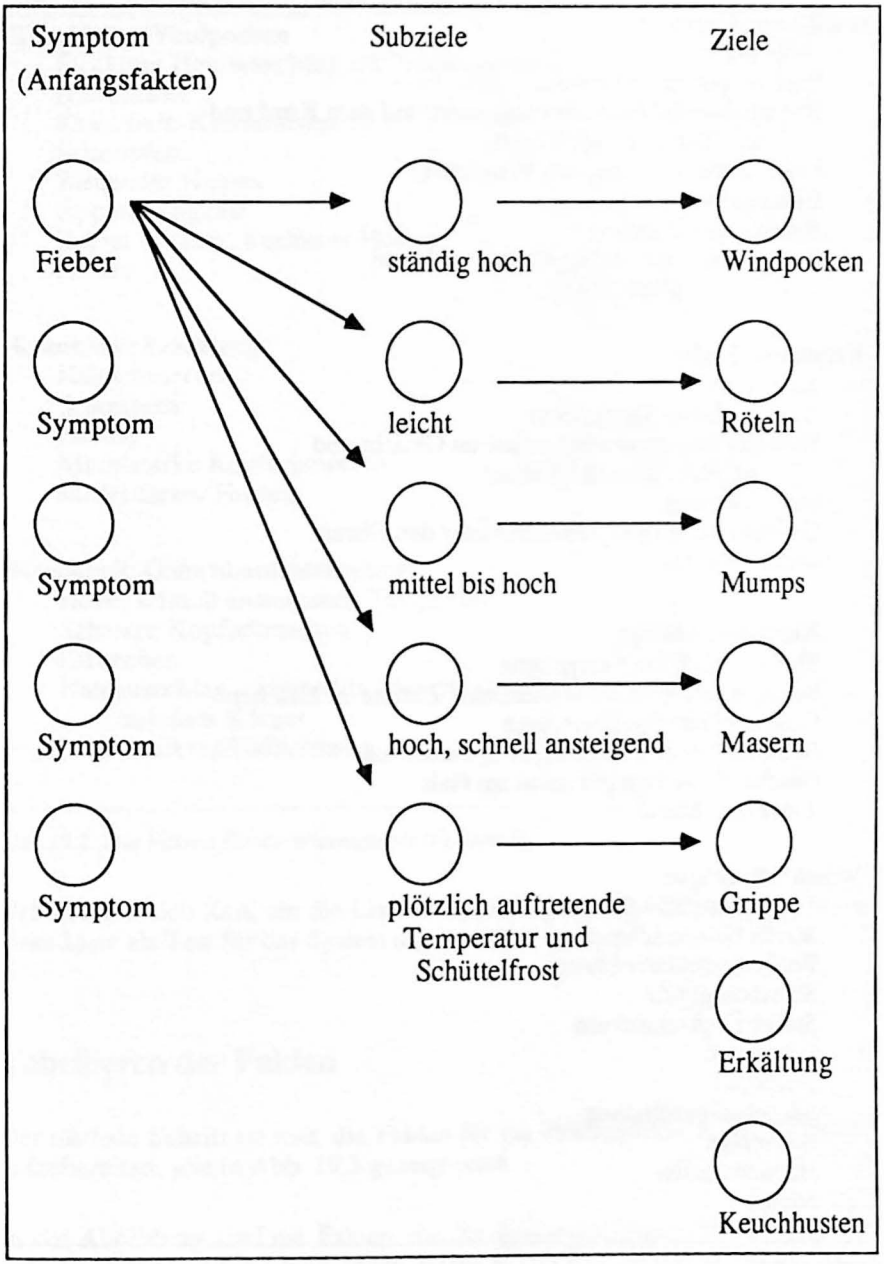


Abb. 19.1: Wege durch den Problemraum

Krankheit: Masern
Schnupfen
Rauher, trockener Husten
Bräunlich-rosa Hautausschlag, zuerst auf dem Kopf und am Hals, dann am Körper
Hohe, schnell ansteigende Temperatur
Blutunterlaufene Augen
Bindehautentzündung
Rote Flecken mit weißen Stellen im Mund
Lichtüberempfindlichkeit

Krankheit: Röteln
Schnupfen
Leicht erhöhte Temperatur
Hellroter Hautausschlag, zuerst im Gesicht und am Hals, dann am Körper
Steife Gelenke
Geschwollene Lymphknoten hinter den Ohren
Halsschmerzen

Krankheit: Mumps
Mittlere bis hohe Temperatur
Schmerzhaft oder unmöglich, eine Zitrone zu lutschen
Geschwollene Speicheldrüsen
Spärlicher oder übermäßiger Speichelfluß
Geschwollene Lymphknoten am Hals
Trockener Mund

Krankheit: Grippe
Temperatur und Schüttelfrost
Starke Gliederschmerzen
Rauher, trockener Husten
Schwächegefühl
Starke Kopfschmerzen
Erbrechen
Durchfall
Bindehautentzündung
Schnupfen
Halsschmerzen
Niesen

Abb. 19.2: Die Fakten für die Wissensbasis

Krankheit: Windpocken
Fleckiger Hautausschlag mit Pickeln und Pusteln
Hautjucken
Krankheit: Keuchhusten
Schnupfen
Ziehender Husten
Appetitlosigkeit
Zuerst leichter, trockener Husten
Niesen

Krankheit: Erkältung
Halsschmerzen
Schnupfen
Niesen
Mittelstarke Kopfschmerzen
Mittelstarker Husten

Krankheit: Gehirnhautentzündung
Hohe, schnell ansteigende Temperatur
Schwere Kopfschmerzen
Erbrechen
Hautausschlag – kleine lila oder rötliche Punkte auf dem Körper
Lichtüberempfindlichkeit

Abb. 19.2: Die Fakten für die Wissensbasis (Fortsetzung)

Nehmen Sie sich Zeit, um die Liste so vollständig wie möglich zu machen, da diese Liste als Test für das System dienen soll.

Tabellieren der Fakten

Der nächste Schritt ist nun, die Fakten für die Wissensbasis in einer Tabelle aufzubereiten, wie in Abb. 19.3 gezeigt wird.

In der Abbildung sind die Fakten, die die Bedingungen darstellen, in Zeilen untereinander und die Folgerungen daraus in Spalten angeordnet. In den meisten Systemen wird diese Tabelle mehrere Seiten umfassen.

Symptome	Krankheiten							
	Masern	Röteln	Mumps	Grippe	Windpocken	Keuchhusten	Erkältung	Gehirnhautentzündung
Schnupfen	√	√		√			√	√
Bräunlich-rosa Ausschlag an Kopf & Hals	√							
Hohe, schnell ansteigende Temperatur	√							√
Rauher, trockener Husten	√			√				
Blutunterlaufene Augen	√							
Bindehautentzündung	√			√				
Weiße Stellen im Mund	√							
Lichtüberempfindlichkeit	√							√
Leicht erhöhte Temperatur		√						
Hellroter Ausschlag in Gesicht und Hals		√						
Leichte Kopfschmerzen								√
Steife Gelenke		√						
Geschwollene Lymphknoten hinter den Ohren		√						
Mittlere bis hohe Temperatur			√					
Unmöglich, Zitrone zu lutschen			√					
Geschwollene Speicheldrüsen			√					
Speichelfluß ist nicht normal			√					
Geschwollene Lymphknoten am Hals			√					
Trockener Mund			√					
Temperatur und Schüttelfrost				√				
Starke Gliederschmerzen				√				
Mittelstarker Husten								√
Schwächegefühl				√				
Starke Kopfschmerzen				√				√
Erbrechen				√				√
Durchfall				√				
Halsschmerzen		√		√			√	
Fleckiger Ausschlag, Pickel & Pusteln					√			
Hautjucken					√			
Erst leichter dann ziehender Husten						√		
Appetitlosigkeit						√		
Niesen			√			√		√
Ausschlag – lila oder rötliche Punkte								√

Abb. 19.3: Tabellieren der Fakten

Symptome	Krankheiten							
	Masern	Röteln	Mumps	Grippe	Windpocken	Keuchhusten	Erkältung	Gehirnhautentzündung
Schnupfen	√	√		√		√	√	
Ausschlag								
– hellrot		√						
– bräunlich-rosa	√							
– fleckig, Pickel, Pusteln					√			
– lila oder rötliche Punkte								√
Fieber								
– leicht	√							
– mittel bis hoch			√					
– hoch, schnell ansteigende	√							√
– plötzliche Temperatur und Schüttelfrost				√				
Husten								
– rauh, trocken	√			√				
– leicht, dann ziehend						√		
– mittelstarke								√
Gliederschmerzen – stark				√				
Kopfschmerzen – stark				√				√
mittelstark								√
Lichtüberempfindlichkeit	√							√
Blutunterlaufene Augen	√							
Bindehautentzündung	√			√				
Weiße Stellen im Mund	√							
Steife Gelenke		√						
Geschwollene Lymphknoten hinter den Ohren		√						
Unmöglich, Zitrone zu lutschen			√					
Geschwollene Speicheldrüsen			√					
Speichelfluß ist nicht normal			√					
Geschwollene Lymphknoten am Hals			√					
Trockener Mund			√					
Schwächegefühl				√				
Erbrechen				√				√
Durchfall				√				
Halsschmerzen		√		√			√	
Hautjucken						√		
Appetitlosigkeit						√		
Niesen				√		√	√	

Abb. 19.4: Gruppieren der Fakten

Bündeln

Der nächste Schritt des Systementwurfs besteht darin, so viele Symptome wie möglich in Gruppen zusammenzufassen. Beispielsweise können verschieden hohe Temperaturen oder unterschiedliche Formen von Hautausschlag gebündelt werden. Dadurch kann zuerst die Frage gestellt werden, ob der Patient unter Hautausschlag leidet, und nur, wenn das der Fall ist, wird nach der Art des Ausschlags weitergefragt (siehe Abb. 19.4). Auf diese Art und Weise rollt eine einzige allgemeine Frage oft mehrere Subziele auf. Die Heuristik eines Systems wird durch die jeweilige Bündelung festgelegt.

Das Gruppieren der Fakten ist unter Umständen ein bißchen schwierig. Viele Expertensysteme widersetzen sich jeglicher Strukturierung, und das Bündeln von Fakten ist im Grunde genommen ein Strukturierungsprozeß. Der Hauptgrund für das Zusammenfassen in Gruppen ist aber das Ermöglichen von höherrangigen Fragen, um den Problemraum zu säubern.

Schreiben des Programms

Sobald die Tabellen fertiggestellt sind, kann man damit beginnen, die Prolog-Regeln aufzuschreiben. Auch hier können wir wieder ausnutzen, daß der Entwurf ein zyklischer Vorgang ist. Es ist durchaus üblich, noch einmal von vorne zu beginnen, die Symptome anders zu bündeln und die Regeln neu zu schreiben.

Dabei ist immer zu beachten, daß die Knoten in einem Problemraum immer zugleich Symptom und Subziel sind. Das endgültige Turbo Prolog-Programm finden Sie in Abb. 19.5 aufgelistet.

```
/* MEDIZINISCHES DIAGNOSE-SYSTEM */
/* FÜR KINDERKRANKHEITEN         */
/* APRIL 1987                    */
/* BEISPIELPROGRAMM - NICHT FÜR  */
/* PROFESSIONELLEN GEBRAUCH      */
/* mit dynamischer Datenbank     */

domains
  krankheit = string
  symptom = symbol
  frage = string
  eingabe = char

database
   xpositiv(symptom,integer)
   xnegativ(symptom,integer)
```

```
/* Verwenden Sie das Menü-Programm der Turbo Prolog-Diskette
aus dem Unterverzeichnis PROGRAMS von Borland International
oder ein ähnliches Menü-Programm */

include"menu.pro"

predicates
    thema(krankheit)
    los
    positiv(frage,symptom)
    negativ(frage,symptom)
    symptom(symptom)
    lösch_fakten
    erinnere(symptom,eingabe)
    frag(frage,symptom,eingabe)
    titel
    einmal
    prozess(symptom,integer)
    diagnose(krankheit)
    schluss(krankheit)

goal
    makewindow(1,7,7,"",0,0,25,80),
    los.

clauses
    los:-
        shiftwindow(1),
        clearwindow,
        einmal,
        write("Möchten Sie noch eine Beratung (j/n)?"),
        readchar(Eingabe),nl,
        Eingabe = 'j',
        los.

    einmal:-
        titel,nl,
        diagnose(_),!,
        save("test.dat"),
        lösch_fakten.

    einmal:-
        write("Bedaure, ich bin nicht in der Lage,"),nl,
        write("die Krankheit zu diagnostizieren."),nl,
        lösch_fakten.

    titel:-
         write("MEDIZINISCHES DIAGNOSE-SYSTEM FÜR
         KINDERKRANKHEITEN"),nl.

    positiv(_,Symptom):-
       xpositiv(Symptom,0),!.
```

```
positiv(Frage,Symptom):-
    not(xnegativ(Symptom,0)),
    frag(Frage,Symptom,Eingabe),
    Eingabe = 'j'.

negativ(_,Symptom):-
    xnegativ(Symptom,0),!.

negativ(Frage,Symptom):-
    not(xpositiv(Symptom,0)),
    frag(Frage,Symptom,Eingabe),
    Eingabe = 'n'.

frag(Frage,Symptom,Eingabe):-
    write(Frage),
    readchar(Eingabe),
    write(Eingabe),nl,
    erinnere(Symptom,Eingabe).

erinnere(Symptom,'j'):-
    asserta(xpositiv(Symptom,0)).

erinnere(Symptom,'n'):-
    asserta(xnegativ(Symptom,0)).

lösch_fakten:-
    retract(xpositiv(_,_)),fail.

lösch_fakten:-
    retract(xnegativ(_,_)),fail.

lösch_fakten.

prozess(Symptom,Wahl):-
    asserta(xpositiv(Symptom,Wahl)).

symptom(Symptom):-
    xpositiv(Symptom,0),!.

symptom(Symptom):-
    xnegativ(Symptom,0),!,fail.

symptom(fieber):-
    positiv("Hat der Patient Fieber (j/n)?",fieber),
    menu(6,40,
    ["Leichtes Fieber",
    "Mittleres bis hohes Fieber",
    "Hohes, schnell ansteigendes Fieber",
    "Plötzliche Temperatur und Schüttelfrost"],
    Wahl),
    Symptom = fieber,
    prozess(Symptom,Wahl).
```

```
symptom(hautausschlag):-
    positiv("Hat der Patient Hautausschlag
     (j/n)?",hautausschlag),
    menu(6,40,
    ["Hellroter Ausschlag, zuerst im Gesicht und am Hals,
     dann am Körper",
     "Bräunlich-rosa Ausschlag, zuerst auf Kopf und Hals,
     dann am Körper",
     "Fleckiger Ausschlag mit Pickeln und Pusteln",
     "Kleine lila oder rötliche Punkte auf dem Körper"],
     Wahl),
     Symptom = hautausschlag,
     prozess(Symptom,Wahl).

symptom(kopfschmerzen):-
    positiv("Hat der Patient Kopfschmerzen
     (j/n)?",kopfschmerzen),
    menu(6,40,
    ["Starke Kopfschmerzen",
     "Mittelstarke Kopfschmerzen"],
     Wahl),
     Symptom = kopfschmerzen,
     prozess(Symptom,Wahl).

symptom(schnupfen):-
    positiv("Hat der Patient Schnupfen
     (j/n)?",schnupfen).

symptom(bindehautentzündung):-
    positiv("Hat der Patient eine Bindehautentzündung
     (j/n)?",bindehautentzündung).

symptom(husten):-
    positiv("Hat der Patient Husten (j/n)?",husten),
    menu(6,40,
    ["Rauher, trockener Husten",
     "Leichter, dann ziehender Husten",
     "Mittelstarker Husten"],
     Wahl),
     Symptom = husten,
     prozess(Symptom,Wahl).

symptom(gliederschmerzen):-
    positiv("Hat der Patient Gliederschmerzen
     (j/n)?",gliederschmerzen).

symptom(halsschmerzen):-
    positiv("Hat der Patient Halsschmerzen
     (j/n)?",halsschmerzen).

symptom(zitrone):-
    positiv("Kann der Patient an einer Zitrone lutschen
     (j/n)?",zitrone).
```

```
symptom(speichel):-
    positiv("Ist der Speichelfluß normal
     (j/n)?",speichel).

symptom(lymphen):-
    positiv("Sind die Lymphknoten am Hals geschwollen
     (j/n)?",lymphen).

symptom(niesen):-
    positiv("Muß der Patient niesen (j/n)?",niesen).

symptom(geschwollene_drüsen):-
    positiv("Hat der Patient geschwollene Speicheldrüsen
     (j/n)?",geschwollene_drüsen).

symptom(blutunterlaufene_augen):-
    positiv("Hat der Patient blutunterlaufene Augen
     (j/n)?",blutunterlaufene_augen).

symptom(weisse_flecken):-
    positiv("Hat der Patient rote Flecken mit weißen
     Stellen im Mund (j/n)?",weisse_flecken).

symptom(gelenke):-
    positiv("Hat der Patient steife Gelenke
     (j/n)?",gelenke).

symptom(ohren):-
    positiv("Sind die Lymphknoten hinter den Ohren
     geschwollen (j/n)?",ohren).

symptom(erbrechen):-
    positiv("Hat der Patient erbrochen (j/n)?",erbrechen).

symptom(durchfall):-
    positiv("Hat der Patient Durchfall (j/n)?",durchfall).

symptom(schwäche):-
    positiv("Hat der Patient ein Schwächegefühl
     (j/n)?",schwäche).

symptom(appetit):-
    negativ("Hat der Patient Appetit (j/n)?",appetit).

symptom(licht):-
    positiv("Sind die Augen des Patienten lichtempfindlich
     (j/n)?",licht).

symptom(mund):-
    positiv("Leidet der Patient unter trockenem Mund
     (j/n)?",mund).

symptom(jucken):-
    positiv("Hat der Patient Hautjucken (j/n)?",jucken).
```

```
diagnose("Masern"):-
      symptom(hautausschlag),
      symptom(fieber),
      symptom(husten),
      symptom(bindehautentzündung),
      symptom(schnupfen),
      symptom(licht),
      thema("Masern").

diagnose("Röteln"):-
      symptom(hautausschlag),
      symptom(fieber),
      symptom(schnupfen),
      symptom(halsschmerzen),
      thema("Röteln").

diagnose("Grippe"):-
      symptom(fieber),
      symptom(kopfschmerzen),
      symptom(gliederschmerzen),
      symptom(bindehautentzündung),
      symptom(husten),
      symptom(schnupfen),
      symptom(niesen),
      thema("Grippe").

diagnose("Erkältung"):-
      symptom(kopfschmerzen),
      symptom(niesen),
      symptom(halsschmerzen),
      symptom(schnupfen),
      symptom(husten),
      thema("Erkältung").

diagnose("Mumps"):-
      symptom(fieber),
      symptom(geschwollene_drüsen),
      symptom(mund),
      thema("Mumps").

diagnose("Windpocken"):-
      symptom(hautausschlag),
      symptom(jucken),
      thema("Windpocken").

diagnose("Keuchhusten"):-
      symptom(husten),
      symptom(niesen),
      symptom(schnupfen),
      thema("Keuchhusten").

diagnose("Gehirnhautentzündung"):-
      symptom(fieber),
      symptom(erbrechen),
```

```
    symptom(kopfschmerzen),
    symptom(hautausschlag),
    symptom(licht),
    thema("Gehirnhautentzündung").

thema("Masern"):-
    xpositiv(fieber,3),
    xpositiv(husten,1),
    xpositiv(hautausschlag,2),
    symptom(blutunterlaufene_augen),
    symptom(weisse_flecken),
    schluss("Masern").

thema("Mumps"):-
    xpositiv(fieber,2),
    symptom(zitrone),
    symptom(speichel),
    symptom(lymphen),
    schluss("Mumps").

thema("Windpocken"):-
    xpositiv(hautausschlag,3),
    schluss("Windpocken").

thema("Röteln"):-
    xpositiv(fieber,1),
    xpositiv(hautausschlag,1),
    symptom(gelenke),
    symptom(ohren),
    schluss("Röteln").

thema("Grippe"):-
    xpositiv(fieber,4),
    xpositiv(kopfschmerzen,1),
    xpositiv(husten,1),
    symptom(erbrechen),
    symptom(durchfall),
    symptom(schwäche),
    schluss("Grippe").

thema("Keuchhusten"):-
    xpositiv(husten,2),
    symptom(appetit),
    schluss("Keuchhusten").

thema("Erkältung"):-
    xpositiv(kopfschmerzen,2),
    xpositiv(husten,3),
    schluss("Erkältung").

thema("Gehirnhautentzündung"):-
    xpositiv(kopfschmerzen,1),
    xpositiv(fieber,3),
    xpositiv(hautausschlag,4),
```

```
        Notfall an."),nl,
        schluss("Gehirnhautentzündung").

    thema(Krankheit):-
        write("Der Patient leidet vermutlich an
        ",Krankheit,"."),nl,
        write("Diese Diagnose wurde nicht von einem Arzt
        gestellt."),nl,
        write("Im Ernstfall sollten Sie einen Arzt
        aufsuchen."),nl.

    schluss(Krankheit):-
        write("Der Patient leidet an ",Krankheit,"."),nl,
        write("Diese Diagnose wurde nicht von einem Arzt
        gestellt."),nl,
        write("Im Ernstfall sollten Sie einen Arzt
        aufsuchen."),nl.
```

Abb. 19.5: Expertensystem für medizinische Diagnosen (Fortsetzung)

Die Programmstruktur ist ähnlich wie die des Programms aus Kapitel 9. Die höchstrangige Regel stellt eine rekursive Schleife dar, die den Rest des Programms für eine oder mehrere Befragungen durchläuft. Das Prädikat *einmal* ist die Haupt-Steuerungsschleife, in der jede mögliche Diagnose überprüft wird, bis eine passende gefunden wird.

Jede Diagnose hat die Form:

```
diagnose(X):-
        symptom(X1),
        symptom(X2),
        .   .   .
        symptom(Xn),
        thema(X).
```

Das Prädikat *symptom* überprüft die Symptome, und wenn die betreffende Symptomliste erfolgreich war, wird eine Diagnose gestellt und das Prädikat *thema* aufgerufen. Dieses Prädikat versucht, die aufgestellte Diagnose zu beweisen. Wenn dieser Versuch scheitert, wird die vermutete Diagnose mit der Meldung "Der Patient leidet vermutlich an XXXX." angezeigt.

Konnte die Krankheit jedoch eindeutig identifiziert werden, dann lautet die Meldung: "Der Patient leidet an XXXX." Durch das Prädikat *thema* kann eine einzige Diagnose im System von mehreren Regeln verwendet werden. Wir wollen nun das Prädikat *symptom* genauer betrachten, da es eine komplexe Funktion im Programm erfüllt. Die Ergebnisse einer Befragung werden in ei-

ner dynamischen Datenbank gespeichert. Zu diesem Zweck wurden zwei Datenbank-Prädikate definiert: *xpositiv* speichert die Fakten, die als wahr bewiesen wurden, und *xnegativ* speichert die Fakten, die als falsch erkannt wurden. Bei dieser grundlegenden Frage ist das zweite Argument immer gleich 0. Muß der Patient beispielsweise niesen, so wird das als

```
xpositiv(niesen,0)
```

gespeichert. Wenn der Patient jedoch nicht niesen muß, so heißt es

```
xnegativ(niesen,0)
```

Wenn nun *symptom(niesen)* aufgerufen wird, unifiziert es zuerst mit

```
symptom(Symptom):-
    xpositiv(Symptom,0),!.
```

und bindet dabei die Variable *Symptom* an den Wert *niesen*. Wenn die Frage zuvor noch nicht gestellt worden ist, so befindet sich *xpositiv(niesen,0)* noch nicht in der Datenbank. Das Rumpfziel wird scheitern, und Backtracking bis zu

```
symptom(Symptom):-
    xnegativ(Symptom,0),!,fail.
```

setzt ein, wobei *Symptom* erneut an *niesen* gebunden wird. Wieder wird das erste Rumpfziel scheitern und Backtracking einsetzen, diesmal vielleicht bis zur Prüfung

```
symptom(niesen):-
    positiv('Muß der Patient niesen (j/n)?',niesen).
```

Das Prädikat *positiv* prüft nach, ob sich noch keine Antwort von einer früheren Frage her in der Datenbank befindet, und stellt dann seinerseits die Frage unter Verwendung des Prädikats *frag*. Dieses Prädikat hat Erfolg, falls es einen Eintrag *xpositiv(Symptom,0)* gibt oder wenn die Frage mit *j* beantwortet wird. Das Prädikat *erinnere* speichert die Antwort in Abhängigkeit von ihrer Aussage in einer von beiden Datenbanken.

Einige der symptom-Prädikate aktivieren ein Menü. Das Prädikat *menu* wird von Turbo Prolog bereitgestellt, und zwar wird es über den Compiler-Befehl *include* geladen. Dieses Prädikat ruft das Menü jedoch nur auf, wenn die Frage mit *j* beantwortet wurde. In diesem Menü kann der Anwender das Symptom gemäß der bereits besprochenen Gruppierung genauer spezifizieren.

Wird z. B. die Frage

```
Hat der Patient Fieber (j/n)?
```

gestellt und der Anwendet antwortet mit *j*, dann erscheint auf der rechten Seite des Bildschirms ein neues Fenster mit den folgenden Auswahlmöglichkeiten:

```
Leichtes Fieber
Mittleres bis hohes Fieber
Hohes, schnell ansteigendes Fieber
Plötzliche Temperatur und Schüttelfrost
```

In diesem Menü erscheint anfangs die erste Auswahlmöglichkeit invers hervorgehoben. Dieser helle Balken kann mit Hilfe der Cursortasten auf die anderen Optionen bewegt werden. Wenn die Auswahl getroffen ist, wird die Wahl durch Drücken der Enter-Taste abgeschlossen. Diese praktische Möglichkeit verhindert Fehleingaben, da nichts eingetippt werden muß. Nachdem die Wahl getroffen ist, verschwindet das Fenster wieder. Falls der Patient kein Fieber hatte, wird das positiv-Prädikat scheitern, und dann erscheint auch kein Menü.

Die Menüwahl wird in der Datenbank gespeichert. Wurde beispielsweise *Mittleres bis hohes Fieber* gewählt, so wird dieser Punkt als

```
xpositiv(fieber,2)
```

gespeichert. Das negativ-Prädikat bewirkt das Gegenteil vom positiv-Prädikat. Wenn die allgemeine Frage nach Fieber verneint wurde, so führt das zum Erfolg des negativ-Prädikats. In der xnegativ-Datenbank wird die Antwort *n* auf die gleiche Art wie beim positiv-Prädikat gespeichert.

Sobald eine Frage beantwortet und die Antwort in einer der beiden Datenbanken gespeichert wurde, wird die Frage im gleichen Programmlauf nicht noch einmal gestellt. Die Prädikate *positiv* und *negativ* forschen immer zuerst in der Datenbank nach, bevor sie eine Frage stellen.

Das Prädikat *diagnose* stellt eine allgemeine Vermutung für eine Diagnose auf, die von dem Prädikat *thema* unter Zuhilfenahme der Menü-Antworten und ein paar anderen symptom-Prädikaten definitiv bewiesen werden soll. Als Ergebnis präsentiert das Programm einen noch unsicheren und einen gesicherten Befund. In dem einen Fall stellt es eine Diagnose auf, die mit hoher Wahrscheinlichkeit zutrifft, und sie wird als erhärtet angenommen, wenn die restlichen Symptome die Diagnose bestätigen.

```
MEDIZINISCHES DIAGNOSE-SYSTEM FÜR KINDERKRANKHEITEN

Hat der Patient Hautausschlag (j/n)?j

        Hellroter Ausschlag, zuerst im Gesicht und am Hals, dann am Körper
        Bräunlich-rosa Ausschlag, zuerst auf Kopf und Hals, dann am Körper
        Fleckiger Ausschlag mit Pickeln und Pusteln
        Kleine lila oder rötliche Punkte auf dem Körper
```

Abb. 19.6: Beginn einer Befragung

```
MEDIZINISCHES DIAGNOSE-SYSTEM FÜR KINDERKRANKHEITEN

Hat der Patient Hautausschlag (j/n)?j
Hat der Patient Fieber (j/n)?j
Hat der Patient Husten (j/n)?n
Hat der Patient Schnupfen (j/n)?n
Hat der Patient Kopfschmerzen (j/n)?j
Hat der Patient Gliederschmerzen (j/n)?n
Muß der Patient niesen (j/n)?n
Hat der Patient geschwollene Speicheldrüsen (j/n)?n
Hat der Patient Hautjucken (j/n)?n
Hat der Patient erbrochen (j/n)?j
Sind die Augen des Patienten lichtempfindlich (j/n)?j
Der Patient leidet vermutlich an Gehirnhautentzündung.
Diese Diagnose wurde nicht von einem Arzt gestellt.
Im Ernstfall sollten Sie einen Arzt aufsuchen.
Möchten Sie noch eine Beratung (j/n)?
```

Abb. 19.7: Eine vollständige Befragung

Am Ende einer Befragung wird die Datenbank von dem Prädikat *lösch_fakten* für einen weiteren Programmlauf gelöscht.

Programmtest

Wenn Sie das Programm eingegeben haben, sollten Sie jede Diagnose im Vergleich mit der ursprünglichen Faktenliste überprüfen. Vergewissern Sie sich, daß die Ergebnisse mit den eingegebenen Fakten übereinstimmen.

Falls es nötig sein sollte, daß Sie Korrekturen im Programm ausführen müssen, ist es ratsam, die korrigierten Prädikate erneut zu überprüfen. Ein Beispiellauf des Programms ist in Abb. 19.6 und in Abb. 19.7 aufgeführt.

Übungen

1. Geben Sie das Programm dieses Kapitels ein, und testen Sie es. Wie gehen Sie beim Testen vor?

2. Bei Gehirnhautentzündung tritt ein Hautausschlag meistens erst ab dem vierten Tag auf. Ändern Sie das Programm so ab, daß es die Krankheit mit und ohne Hautausschlag erkennt.

3. Worin liegen die Stärken und die Schwächen des Programms?

Kapitel 20

Abenteuerspiele und natürliche Abfragesprachen

Der Entwurf von Spielen und speziell von Abenteuerspielen stellt in der *künstlichen Intelligenz* eine besonders reizvolle Aufgabe dar und kann als besondere Form eines Experimentes aufgefaßt werden. Künstliche Intelligenz ist ein Teilbereich der Informatik, der sich mit den Begriffen und Methoden symbolischer Schlußfolgerungen durch einen Computer und der symbolischen Darstellung von Wissen durch Schlußfolgerungs-Mechanismen beschäftigt. (Expertensysteme bilden ein Teilgebiet der künstlichen Intelligenz.)

Das erste programmierte Abenteuerspiel, *Adventure 350*, stellte die ersten Gehversuche im Bereich der künstlichen Intelligenz dar. Die Programmierung von Abenteuerspielen liefert einen guten Einstieg in das Wissensgebiet der künstlichen Intelligenz und der symbolischen Schlußfolgerungen. Zudem sind Abenteuerspiele als Versuchsfeld für den Einsatz von *natürlichen Abfragesprachen* sehr geeignet. Es ist anzustreben, daß der Spieler seine Eingaben in normalem deutschen Sprachgebrauch tätigen kann und trotzdem vom Programm verstanden wird.

Da Turbo Prolog eine symbolische Sprache ist, eignet es sich besonders gut zum Programmieren von Abenteuerspielen. Seine Schlußfolgerungs- und symbolischen Verarbeitungsmöglichkeiten vereinfachen die Handhabung von Objekten (Menschen, Dinge, Gegner usw.) aufgrund von formalen Regeln.

Mit Turbo Prolog können wesentlich komplexere Spiele als mit irgendeiner prozeduralen Sprache erstellt werden. Zudem muß für das Programmieren in Turbo Prolog weniger Zeit als bei prozeduralen Sprachen aufgewendet werden.

Der Hauptvorteil von Prolog gegenüber anderen Sprachen besteht bei der Verarbeitung von natürlichen Sprachmöglichkeiten darin, daß es eine symbolische Sprachverarbeitung durchführt.

Im Gegensatz zu anderen Sprachen können in Prolog Wörter symbolisch dargestellt werden, und die Beziehungen der Symbole untereinander können in

formalen Regeln ausgedrückt werden. Das vorliegende Kapitel zeigt ein paar allgemeine Richtlinien für den Entwurf von Abenteuerspielen unter Einbeziehung einer natürlich-sprachlichen Schnittstelle.

Schließlich wird ein natürlich-sprachlicher Interpreter vorgestellt, der in Turbo Prolog geschrieben wurde und Sie in Ihren Bemühungen, ein eigenes Spiel zu entwerfen, unterstützen wird.

Entwurf eines Geheimnisses

Abenteuerspiele werden um ein Geheimnis herum gesponnen und durch die Aktionen von Forschern und Abenteurern belebt. Den Entwurf eines Spiels sollten Sie damit beginnen, das zu erreichende Ziel festzulegen. Sie müssen sich eine Geschichte ausdenken und dabei sowohl die Welt, in der sie spielen soll, als auch die Geschichte selbst erfinden. Sie sollten außerdem Zwischenziele definieren, die der Forscher auf verschiedenen Spielebenen erreichen kann.

Wenn Sie das Spiel zu schwierig gestalten, ohne zwischenzeitliche Erfolgserlebnisse vor dem eigentlichen Ende, kann es passieren, daß der Forscher das Interesse verliert. Recherchieren Sie die Fakten für die Geschichte und den Schauplatz. Wenn die Geschichte auf einem historischen Ereignis oder einer Legende basiert, sollten die Fakten genau wiedergegeben werden. Sorgen Sie dafür, daß die Geschichte einfallsreich und unterhaltsam ist, und bauen Sie etwas Humor und ein paar Tricks ein, um den Spieler zu erheitern. Machen Sie die Umgebung in hohem Maße kalkulierbar.

Der Entwurf einer Fantasiewelt macht im allgemeinen den größten Zeitfaktor bei der Erstellung eines guten Abenteuerspiels aus, da man dafür viel Kreativität und Vorstellungskraft einsetzen muß. Die Beschaffenheit dieser Welt sollte den Vorstellungen des Entdeckers entgegenkommen. Beispielsweise sollte ein Mond-Reisender nicht ohne Raumanzug herumlaufen.

Es können durchaus außergewöhnliche Ereignisse auftreten oder fremdartige Umgebungen erscheinen, nur sollte es für alles, was passiert, eine logische Erklärung geben. Witzige Situationen und Fingerzeige sollten nicht zu sparsam eingesetzt werden, und die Geschichte sollte interessant und abwechslungsreich sein und zwischendurch Erfolgserlebnisse vermitteln. Vermeiden Sie es, eine geradlinige Lösung oder auch nur den Anschein einer Lösung ohne Umwege zu geben. Für den Spieler sollte sich das Spiel als endlos darstellen, auch wenn es nur eine einzige Möglichkeit gibt, das Spiel zu beenden. Bauen Sie Fallen, Gefahren und feindselige Elemente ein.

Sie sollten einen Plan zeichnen, auf dem Sie die Räume Ihrer Fantasiewelt und in jedem Raum die Objekte einzeichnen. Ein Beispielplan ist in Abb. 20.1 zu sehen. Dabei handelt es sich nur um den Teil eines Spielplans, aber er kann einen Eindruck darüber vermitteln, wie ein solcher Plan aussehen könnte. Jeder Raum hat eine eindeutige Bezeichnung und steht mit einem oder mehreren anderen Räumen in Verbindung. Ein solcher Raumverbund kann in Prolog wie folgt deklariert werden:

```
raum(Raumnr,N,S,O,W,NO,NW,SO,SW,AB,AUF)
```

Dabei haben die Argumente die folgenden Bedeutungen:

Raumnr	*Raumnummer*
N, S, O, W	Raumnummer des Raums, der in dieser Richtung liegt (0, wenn es keinen benachbarten Raum gibt)

Bei der höchstrangigen Regel handelt es sich wieder um eine einfache Rekursionsschleife:

```
los:-
    repeat,
    (Ausgabe von einführenden Erläuterungen)
    einmal.

los:-
    write("Vielen Dank fürs Spielen.").

einmal:-
    write(">"),
    readln(Eingabe),
    upper_lower(Antwort,Eingabe),
    Antwort <>"ENDE",
    mach(Antwort),
    einmal.
```

Erster Spielentwurf

Die Programmausführung ist ein Prozeß, der immer wieder von vorn beginnt und an dem ein Eingabe-Prädikat und drei Hauptmodule beteiligt sind (siehe Abb. 20.2). Das Modul zur Syntaxanalyse, im allgemeinen *Parser* genannt, ist die natürlich-sprachliche Schnittstelle. Das Semantik-Modul analysiert die vom Parser zergliederte Ausgabe und führt die vom Anwender geforderten Operationen aus. Das Ereignis-Modul aktiviert die je nach Spielstand notwendigen Aktionen. Wir wollen in diesem Kapitel eine ausführliche Erläuterung des Parsers oder der natürlich-sprachlichen Schnittstelle geben. Dem Leser bleibt die Entwicklung eines Semantik- und eines Ereignis-Moduls selbst überlassen.

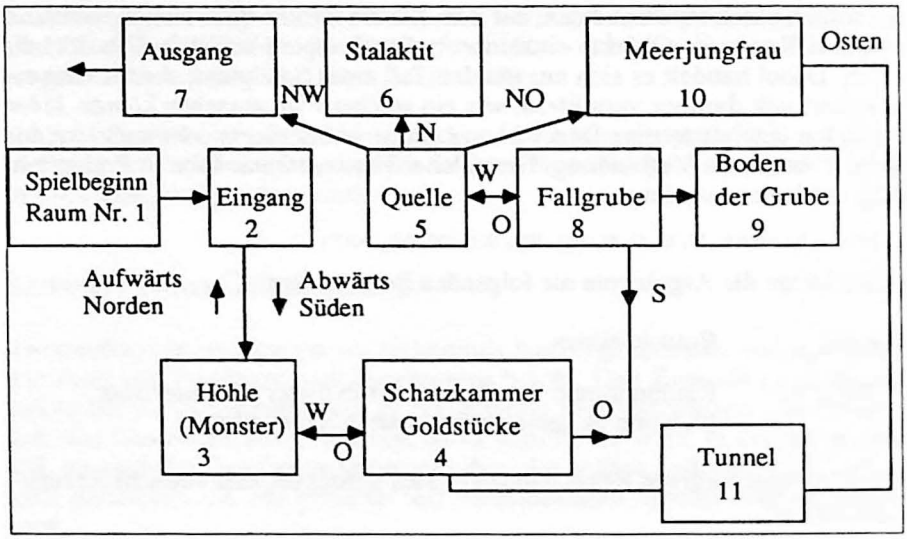


Abb. 20. 1: Ein Abenteuerplan

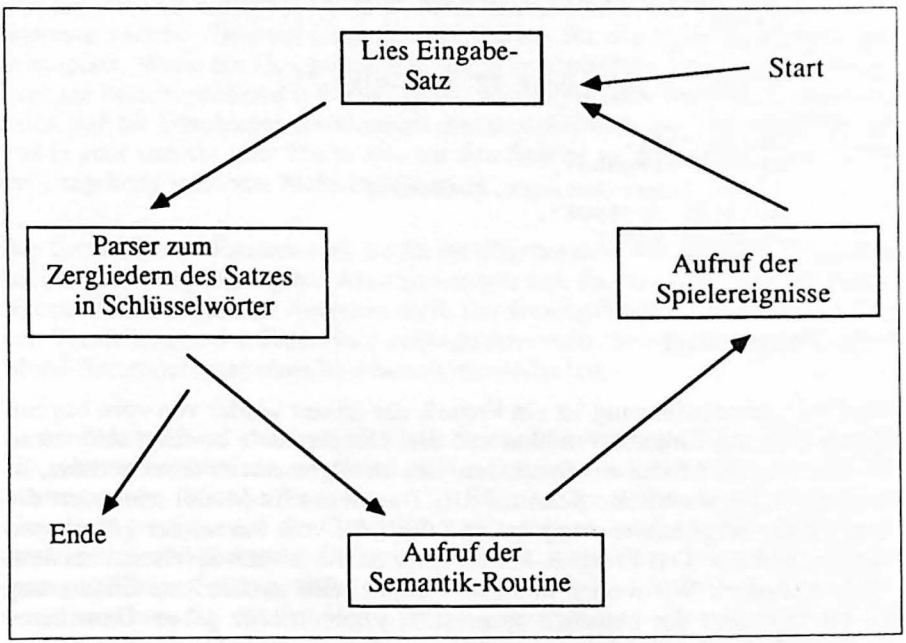


Abb. 20.2 Der Spielkreislauf

Prolog und natürliche Sprachverarbeitung

In Turbo Prolog kann eine natürlich-sprachliche Schnittstelle definiert werden, die es dem Anwender ermöglicht, das Spiel mit normalen deutschen Sätzen zu steuern.

Ein Programmsystem, das mit einem solchen natürlich-sprachlichen Prozessor betrieben wird (wie z. B. ein Abenteuerspiel), nennt man auch *NLP-gesteuertes System* (NLP– Natural Language Processor). In einem derartigen System kennt der NLP zumeist nur eine Untermenge der gesamten natürlichen Sprache. Das bringt in den meisten Fällen für den Anwender aber nur geringfügige Einschränkungen mit sich. Ein gutes Abenteuerspiel erkennt möglicherweise zwischen ein paar hundert bis weit über tausend Wörter.

Man kann davon ausgehen, daß ein Eingabesatz als Kernstück immer ein Subjekt (Nomen) und ein Verb enthält. Bei weiter fortgeschrittenen Anwendungen werden sogar Objekte, Adverben, Adjektive und andere Wortarten erkannt. Die Satzstellung kann dabei, abhängig von der Funktionsweise des NLP, von Bedeutung sein oder nicht. Der Grundwortschatz befindet sich im allgemeinen in einer dynamischen Datenbank, die mit Hilfe des Prädikats *consult* in das System geladen wird.

Ein guter Ausgangspunkt für eine solche Schnittstelle ist SANAL, ein Satz-Analysierprogramm, das Sie auf einer der drei Turbo Prolog-Disketten im Unterverzeichnis *EXAMPLES* finden können. Das Prinzip beim Entwurf einer natürlich-sprachlichen Schnittstelle besteht darin, den vom Anwender eingegebenen Satz grammatikalisch zu zergliedern, wobei das Prädikat *fronttoken* den Satz in eine Liste von Wörtern aufschlüsselt, und zwar auf folgende Weise:

```
aufteilen(Satz,[Kopf|Rumpf]):-
    fronttoken(Satz,Kopf,S1),,
    aufteilen(S1,Rumpf).

aufteilen(_,[]).
```

Danach können die Wörter des Eingabesatzes mit Hilfe eines Wörterbuchs in Nomen, Verben, Adjektive und Adverben eingeteilt werden.

Sobald der Satz zergliedert ist, werden die Schlüsselwörter dazu verwendet, *Semantik-Prädikate* zu aktivieren. In einem solchen Programm sind mehrere Semantik-Prädikate notwendig, und zwar eins für jede Gruppe von Verben. Diese Semantik-Prädikate leisten die eigentliche Arbeit. Sie können beispielsweise den Spieler von einem Raum in den anderen bringen oder ihn

Nach dem Aufruf des Semantik-Prädikats wird von einem *Ereignis-Prädikat* geprüft, welche Aktionen abhängig von den voreingestellten Bedingungen zu starten sind. Beispielsweise kann das Ereignis-Prädikat prüfen, ob das Licht des Abenteurers erloschen ist oder ob er in eine Fallgrube geraten ist oder gar von einem Feind getötet wurde. Es könnte vielleicht nachprüfen, ob ein Gegner auf der Bildfläche erschienen ist, oder die Reserveanzeige für die Batterie oder den Ölverbrauch der Lampe des Forschers neu einstellen (womöglich ist die Batterie oder das Öl zur Neige gegangen). Vielleicht ist der Forscher auch hungrig, oder er braucht etwas Schlaf.

Wenn das Ereignis-Prädikat seine Aufgabe erfüllt hat, beginnt der Kreislauf von vorne, und Prolog fordert eine weitere Benutzereingabe.

Die folgenden Fakten sollten in jedem guten Spielentwurf berücksichtigt werden:

- Der normale Eingabesatz enthält ein Nomen und ein Verb, z. B.:

  ```
  Sie sind in einem Raum mit einer hohen Decke.
  Auf dem Boden liegen Goldstücke.
  >NIMM GOLD
  ```

  oder

  ```
  >GEHE WESTEN
  ```

  Für ein einfaches Abenteuerspiel reicht es aus, wenn Nomen und Verben erkannt werden.

- Durch Verwendung von Adjektiven oder Präpositionen können zwei ähnliche Gegenstände in einem Raum voneinander unterschieden werden, zum Beispiel:

  ```
  >ÖFFNE DIE HÖLZERNE TÜR
  Die hölzerne Tür ist jetzt offen.
  In Ihrer Nähe befindet sich eine offene Kiste.
  In der Kiste ist ein goldenes Halsband.
  >NIMM DAS GOLDENE HALSBAND AUS DER KISTE
  OK
  >
  ```

  Wenn das Adjektiv HÖLZERN weggelassen wird, stellt das Programm dem Spieler eine genauere Frage:

```
>ÖFFNE DIE TÜR
Welche Tür?
>ÖFFNE DIE HÖLZERNE TÜR
.  .  .
```

- Manchmal ist es nicht notwendig, daß das Verb mit einem Nomen zusammen verwendet wird. Wenn beispielsweise der Anwender die Eingabe UNTERSUCHE eintippt und der Spieler gerade einen Gegenstand in der Hand hält, sollte Prolog wissen, daß dies der Gegenstand ist, der untersucht werden soll.

- Einige Himmelsrichtungen (wie etwa SÜDEN oder S) werden als Nomen behandelt und können entweder von einem Verb begleitet werden oder nicht, z. B.:

```
>SÜDEN
>GEHE SÜDEN
```

- Sie können auch gewisse Verben zusammen mit bestimmten Nomen als überflüssig weglassen. Dann bedeutet das Wort GOLD in einem Raum, in dem sich Gold befindet, beispielsweise das gleiche wie NIMM GOLD.

- Einige Kombinationen von Verben und Nomen sind nicht erlaubt. Die Zulässigkeit einer bestimmten Kombination muß geprüft werden.

- Das Programm sollte Wörter mit gleichen Bedeutungen erkennen. Beispielsweise könnte LAUFE das gleiche bewirken wie GEHE.

- In manchen Fällen kann ein Objekt wieder andere Objekte enthalten. Eine Truhe enthält zum Beispiel eine Halskette oder eine Streichholzschachtel ein Streichholz. In Turbo Prolog können sehr komplexe hierarchische Strukturen gebildet werden. Die Truhe kann eine Liste von Objekten enthalten und jedes davon wieder eine Liste von anderen Objekten. Oder ein Objekt kann in einer zusammengesetzten Struktur mit einem Adjektiv verbunden werden, so wie *tür(hölzern)*.

- Ein spezielles Objekt kann sowohl statische als auch dynamische Eigenschaften besitzen. Ein statisches Merkmal wird sich während des ganzen Programmlaufs nicht verändern. Statische Merkmale umfassen die Ausgangsposition eines Objekts, bestimmte Eigenschaften des Objekts, wie z.B. Gewicht und Wert, und möglicherweise auch Verwendungshinweise, die besagen, ob ein Objekt gegessen, verbrannt oder ob damit gekämpft werden kann. Dynamische Eigenschaften können sich während des Spiels verändern. Dazu gehört die aktuelle Position des Objekts und der derzeitige Zustand (offen oder geschlossen, zerstört oder unzerstört).

- Ortsangaben können manchmal lange und manchmal kurze Beschreibungen erfordern. Wird ein Raum zum erstenmal betreten, kann die lange Beschreibung angezeigt werden. Im weiteren Verlauf genügt es, wenn beim Betreten desselben Raums eine kürzere Beschreibung gegeben wird. Das bedeutet, daß Prolog eine Liste der bereits aufgesuchten Räume führen und bei jeder Ortsveränderung überprüfen muß, ob dieser Raum bereits betreten wurde.
- Das Ereignis-Prädikat kann auch *Hintergrund*-Prädikat genannt werden, da es Ereignisse in der Fantasiewelt ohne Eingaben des Spielers ablaufen läßt. In einem weiterentwickelten Abenteuerspiel könnte ein zweites Ereignis-Prädikat vor Ablauf der Semantik-Routine aufgerufen werden.

Kreativität mit Turbo Prolog

Die meisten Spiele verfügen über einen formalen Regelsatz, der den Spielablauf bestimmt. Um zu gewinnen, braucht man vielleicht Glück oder Geschick oder eine Kombination von beidem. Wenn der Spieler aber ein Abenteuerspiel gewinnen will, muß er eine Strategie oder eine Heuristik entwickeln, die ihn schneller ans Ziel bringt als jeden anderen.

Da Prologs Stärke ein symbolischer Schlußfolgerungs-Mechanismus ist, stellt es die ideale Sprache zum Entwickeln von Spielen dar. Ein Programm kann so entworfen werden, daß es als Herausforderer Ihrer eigenen Taktik und Kreativität auftritt. Andererseits können Sie aber auch Regeln in ein Prolog-Programm einbauen, die Ihnen Hilfestellungen beim Entwickeln einer Gewinnstrategie anbieten.

Mit ein wenig Erfahrung werden Sie bald in der Lage sein, Spiele, die sich einfacher Schlußfolgerungs-Mechanismen bedienen, selbst in Prolog zu programmieren, wie z. B. Superhirn, Nimm, Kalab und die Türme von Hanoi. Beispiele für diese Spiele finden Sie in vielen Prolog-Büchern. Bei umfangreicheren Spielen, wie etwa das Problem der N Damen aus dem Turbo Prolog-Handbuch, können Sie damit beginnen, einen Teilbereich zu definieren und diesen in Turbo Prolog zu programmieren.

Ein Beispiel

Wenn Sie Lust haben, Ihre Kreativität bei der Erstellung eines eigenen Abenteuerspiels in Turbo Prolog auszuprobieren, können Sie das Programm aus Abb. 20.3 als Grundlage dafür nehmen. In diesem Beispiel kommt eine natürlich-sprachliche Schnittstelle zum Einsatz, die dem Anwender erlaubt, den Dialog mit dem Spiel in ganzen Sätzen zu führen.

```
/* BEISPIEL FÜR DEN ENTWURF VON PARSER-
PRÄDIKATEN FÜR EIN ABENTEUERSPIEL */

domains
    spalte,nr = integer
    wort,text,schlüssel = string
    nomen,verb = reference string
    n,s,o,w,no,nw,so,sw,ab,auf = integer
    bewegliste = string*
    wortliste = wort*

predicates
    los
    einmal
    tok(spalte,text,nomen,verb)
    prüfnomen(schlüssel,nomen)
    prüfverb(schlüssel,verb)
    ist_Nomen(schlüssel)
    ist_Verb(schlüssel)
    parser(text,nomen,verb)
    mitglied(wort,wortliste)
    nomen(schlüssel)
    verb(schlüssel)
    repeat
    synverb(wort,wort)
    prüfsyn(wort,wortliste,wort)
    kopf_von(wortliste,wort)

/* An dieser Stelle müssen die Räume und Pläne in
der Datenbank gespeichert werden. */
/* Die Angaben für einen Raum werden in Prädikaten von
folgender Form gespeichert: */
/* raum(nr,n,s,o,w,no,nw,so,sw,ab,auf) */
/* Dabei gibt beispielsweise n die Raumnummer an,
die in Richtung Norden liegt. Wenn kein Raum in
dieser Richtung liegt, nimmt n den Wert 0 an. */

/* beschreib(nr) gibt die Raumbeschreibung für den
Raum mit der Nummer nr aus */

goal
    makewindow(1,7,7,"ABENTEUERSPIEL",0,0,25,80),
    los.

clauses
    los:-
        repeat,
        clearwindow,
        write("Sie befinden sich am Eingang einer großen
           Höhle."),nl,
        einmal,
        write("Möchten Sie noch einmal spielen (j/n)?"),
     readchar(Eingabe),
        Eingabe = 'n'.
```

Abb. 20.3: Ein einfacher Parser für ein Abenteuerspiel

```
    los.

    einmal:-
        write("> "),
        readln(Text),
        upper_lower(änderText,Text),
        änderText <> "ENDE",!,
        parser(änderText,_,_),
        /* parser gibt nomen und verb gebunden zurück. */
        /* Fügen Sie hier die Syntaxanalyse ein. */
        /* Fügen Sie hier das Ereignis-Prädikat ein. */
        einmal.

    einmal.

    repeat.

    repeat:-
        repeat.

/* Hier beginnt die Parser-Routine, die das
Nomen und das Verb heraussucht */

    parser(Text,Nomen,Verb):-
        tok(3,Text,Nomen,Verb),
        bound(Nomen),
        bound(Verb),
        Wort = Verb,
        synverb(Wort,AusVerb),
        write("Nomen = ",Nomen),nl,
        write("Verb = ",Verb," umgewandelt zu: ",Aus
        Verb),nl,!.

    parser(_,_,_):-
        write("Das verstehe ich leider nicht."),nl.

/* Die folgende Routine sucht das Synonym für ein Wort.
Das Wort wird mit einer Liste verglichen, und das erste
Wort der Liste wird zurückgegeben. Im folgenden Beispiel
werden alle Wörter, die eine Bewegung ausdrücken, in das
Wort GEHE geändert. Bei einem richtigen Spiel ist hier eine
Liste von Listen erforderlich. Das Eingabewort wird mit jeder
Liste verglichen. Wenn das Wort in einer Liste gefunden
wird, wird das erste Wort dieser Liste zurückgegeben. */

    synverb(Wort,Auswort):-
        Y = ["GEHE","LAUFE","SCHLEICHE","MAR-
        SCHIERERE","RENNE","DREHE"],
        prüfsyn(Wort,Y,Auswort).

    prüfsyn(Wort,Y,Auswort):-
        mitglied(Wort,Y),
        kopf_von(Y,Auswort).
```

Abb. 20.3: Ein einfacher Parser für ein Abenteuerspiel (Fortsetzung)

```
    prüfsyn(Wort,_,Auswort):-
        Auswort = Wort.

    kopf_von([Kopf|_],Kopf).

/* Jetzt wird der Eingabesatz eingelesen und
zergliedert. Dabei werden das Nomen und das Verb
herausgefiltert. */

    tok(Spalte,Text,Nomen,Verb):-
        fronttoken(Text,Schlüssel,Textx),
        prüfnomen(Schlüssel,Nomen),
        prüfverb(Schlüssel,Verb),
        !,
        str_len(Schlüssel,Länge),
        Pos1 = Spalte + Länge + 1,
        tok(Pos1,Textx,Nomen,Verb).

    tok(_,_,_,_).

    prüfnomen(Schlüssel,Nomen):-
        ist_Nomen(Schlüssel),
        Nomen = Schlüssel,!.

    prüfnomen(_,_).

    prüfverb(Schlüssel,Verb):-
        ist_Verb(Schlüssel),
        Verb = Schlüssel,!.

    prüfverb(_,_).

/* Die folgenden Routinen prüfen, ob das Wort
ein Nomen oder ein Verb ist. */

    ist_Nomen(Schlüssel):-
        nomen(Schlüssel).

    ist_Verb(Schlüssel):-
        verb(Schlüssel).

    mitglied(X,[X|_]).

    mitglied(X,[_|Y]):-
        mitglied(X,Y).

/* Nun folgt das Wörterbuch, das Sie auch
als dynamische Datenbank führen können. */

    nomen("NORDEN").
    nomen("SÜDEN").
    nomen("OSTEN").
    nomen("WESTEN").
```

Abb. 20.3: Ein einfacher Parser für ein Abenteuerspiel (Fortsetzung)

```
verb("GEHE").
verb("LAUFE").
verb("SCHLEICHE").
verb("MARSCHIERE").
verb("RENNE").
verb("DREHE").
```

Abb. 20.3: Ein einfacher Parser für ein Abenteuerspiel (Fortsetzung)

Wenn das Spiel gestartet wird, erscheint eine Nachricht darüber, wo Sie sich befinden. Dann wartet das Programm auf einen Eingabesatz. Geben Sie einmal den folgenden Satz ein:

```
Gehe nach Westen.
```

Im Programm läuft folgendes ab:

1. Alle Zeichen des Satzes werden in Großbuchstaben umgewandelt. Sofern die Eingabe nicht aus dem Wort ENDE besteht, ruft das Programm das Prädikat *parser* auf, um das zu prüfende Verb (GEHE) und das Nomen (WESTEN) in dem Satz zu finden. (Beachten Sie, daß die Richtungsangaben als Nomen behandelt werden und in einem Wörterbuch beliebige Wortarten verwendet werden dürfen.)

2. Das Prädikat *tok* durchsucht dann den Satz ab Spalte 3, um das Verb und das Nomen zu finden. Jedes Token (Wort) wird mit dem Wörterbuch der Nomen und der Verben verglichen. Wenn eine Übereinstimmung gefunden wird, wird je nachdem die Variable *Nomen* oder die Variable *Verb* an das Suchwort gebunden.

3. Die Semantik-Routinen (hier nicht aufgeführt) erkennen nur ein paar Wörter. Beispielsweise sollte eine Semantik-Routine für Vorwärtsbewegungen zulassen, daß die Wörter GEHE, LAUFE, RENNE, KLETTERE als Synonyme gebraucht werden können, weil sie im Prinzip das gleiche ausdrücken. Ein einziges Semantik-Prädikat der Form

   ```
   semantik("GEHE",Nomen,AlteStelle,NeueStelle)
   ```

   sollte dafür ausreichen. Aus diesem Grund wurden alle Verben, die eine Bewegung ausdrücken, in einer Liste zusammengefaßt (betrachten Sie das Prädikat *synverb*), und jedes Wort, das in der Liste gefunden wird, wird in das erste Wort dieser Liste (GEHE) umgewandelt.

Nachdem das Nomen und das Verb gefunden und ausgegeben wurden, beginnt das Programm mit einer weiteren Eingabeaufforderung von vorn. Durch das

Hinzufügen von Semantik- und Ereignis-Routinen wird aus dem Programm ein vollständiges Spiel. Außerdem fehlen noch mehrere Listen mit Synonymen und schließlich eine Liste der Listen, um das endgültige Nomen und das Verb zu erhalten.

Einen weiteren Gesichtspunkt sollten Sie bei dem vorliegenden Programm noch berücksichtigen. Die Objekte *Nomen* und *Verb* wurden im Abschnitt *domains* als Objekte vom Typ *reference* deklariert. Eine solche Deklaration ist erforderlich, wenn ungebundene Variablen von einem Subziel an ein anderes übergeben werden. Wenn Sie diese nicht als Objekte vom Typ *reference* auszeichnen, wird bei der Compilierung eine Warnung an der Stelle ausgegeben, wo zum zweitenmal die Prädikate *prüfnomen* und *prüfverb* auftreten. Das Programm wird zwar trotzdem laufen; man kann sich aber dadurch eine Warnmeldung ersparen.

Übungen

1. Tippen Sie das Programm aus Abb. 20.3 ein, und compilieren Sie es. Was passiert, wenn Sie die folgenden Eingabesätze eintippen? (Nennen Sie die voraussichtliche Ausgabe von Nomen und Verb.)

a. Ich glaube, ich gehe nach Osten.

b. Was passiert, wenn ich nach Westen laufe?

c. Springe nach Süden.

d. Gehe nach Hause.

2. Ändern Sie das Beispielprogramm so ab, daß die Nomen- und Verben-Wörterbücher in einer dynamischen Datenbank gespeichert werden. Worin liegen dabei die Vor- und die Nachteile?

3. Fügen Sie eine zweite Synonymliste *["N","NORDEN"]* für das Nomen im Prädikat *prüfnomen* hinzu. Erstellen Sie ähnliche Listen für die anderen Richtungen. Fassen Sie diese Listen in einer übergeordneten Liste zusammen, und erstellen Sie ein Prädikat, das die Nomen und Verben in die jeweils ersten Elemente der Liste (Kopf) umwandelt, in der sie enthalten sind.

4. Geben Sie eine Semantik-Routine für Bewegungen ein:

```
semantik("GEHE",Nomen,AlteStelle,NeueStelle)
```

die den Spieler von einem Raum in den nächsten bringt. Fügen Sie fünf weitere Räume hinzu. um die Routine zu testen.

Anhang A

Installierung von Turbo Prolog

Bevor Sie zum erstenmal mit Turbo Prolog arbeiten, sollten Sie ein paar einfache Dinge tun. In diesem Anhang erläutern wir Schritt für Schritt, wie Sie vorgehen sollen. Zunächst geben wir eine kurze Zusammenfassung:

1. Legen Sie zunächst alles bereit, was Sie brauchen. Auf die notwendige Hardware und Software werden wir später in diesem Kapitel zu sprechen kommen.

2. Fertigen Sie Kopien der Turbo Prolog-Disketten an. Beschriften Sie die Kopien, verwahren Sie die Originaldisketten an einem sicheren Ort, und verwenden Sie für alle weiteren Arbeiten nur noch die Kopien.

3. Lesen Sie sich die README-Datei durch, um herauszufinden, welche Änderungen für Ihre speziellen Anwendungen gemacht werden müssen.

4. Wenn Sie über eine Festplatte verfügen, sollten Sie die Programme darauf installieren. Falls Sie keine Festplatte haben, sollten Sie eine Arbeitsdiskette erstellen, die nur die wichtigsten Dateien zum Arbeiten mit Prolog enthält.

5. Verschaffen Sie sich einen Überblick über den Inhalt der Turbo Prolog-Disketten.

Was Sie brauchen

Wir wollen nun betrachten, was wir alles brauchen, um mit Turbo Prolog zu arbeiten.

Hardware

Zunächst einmal benötigen Sie einen IBM PC-kompatiblen Computer z.B. einen IBM-AT oder einen gleichartigen Rechner. Wenn möglich, sollte Ihnen

eine Festplatte zur Unterstützung der Arbeit mit Turbo Prolog zur Verfügung stehen. Die Geschwindigkeit einer Festplatte und der Komfort, den sie bietet, machen sich bezahlt. Wenn Sie keine Festplatte verwenden, sollten Sie zumindest zwei Diskettenlaufwerke zur Verfügung haben, um allzu häufiges Wechseln der Disketten zu vermeiden.

Ihr Computer muß mindestens mit 386 KByte Speicher ausgestattet sein. Dies ist bei allen neueren Geräten ohnehin gegeben, nur ältere PCs oder Original-IBM-PCs wurden in größeren Stückzahlen mit kleineren Speichern ausgeliefert. Diese Rechner müssen Sie zuvor mit einer Speichererweiterungskarte aufrüsten. Tatsächlich werden Sie feststellen, daß für sinnvolle Anwendungen mindestens 640 KByte Speicherkapazität notwendig sind, da sich Turbo Prolog-Programme überwiegend im Speicher befinden. Auch einige Beispielprogramme in diesem Buch benötigen mehr als das Minimum von 386KB (512 KByte sind aber ausreichend). Es gibt zwar ein paar Möglichkeiten, um mit Hilfe von Dateien die Speicheranforderungen in Turbo Prolog für bestimmte Anwendungen zu reduzieren, aber Sie werden feststellen, daß es bei der Programmierung in Prolog aus Geschwindigkeitsgründen besser ist, alle Datem im Speicher zu halten.

Um so viel Speicherplatz wie möglich zur Verfügung zu haben, sollten Sie es vermeiden, Turbo Prolog zusammen mit bestimmten lokalen Netzwerkprodukten zu betreiben (die ersten Versionen der Netzwerk-Software von IBM verbrauchen zuviel Speicherplatz und lassen unter DOS 3.2 und älteren DOS-Versionen zuwenig Platz für andere Software übrig). Achten Sie darauf, daß andere speicherresidente Programme, wie z. B. SideKick und ProKey, die den für Prolog verfügbaren Speicherplatz verringern, aus dem Speicher entfernt werden.

Mit Turbo Prolog kann fast jede Grafikkarte betrieben werden. Mit der Fenstertechnik wird es keine Probleme geben, und Sie können jedes beliebige Prolog-Programm erstellen, verändern und ausführen, solange Sie die Grafik-Prädikate nicht verwenden. Wenn Sie allerdings die Grafik-Möglichkeiten von Turbo Prolog ausnutzen wollen, benötigen Sie entweder eine CGA-Farbgrafikkarte (Color Graphic Adaptor) oder eine erweiterte Grafikkarte (Extended Graphic Adaptor – EGA) bzw. eine Karte, die mindestens einen dieser Modi beherrscht. Die Grafik-Prädikate funktionieren beispielsweise nicht bei der Hercules-Karte.

Außerdem werden Sie für Ihr System einen Drucker brauchen. Turbo Prolog ist nicht auf einen bestimmten Druckertyp festgelegt. Es kann fast jeder Drukker verwendet werden. Wenn Sie allerdings Hardcopies anfertigen wollen (das sind Kopien vom Bildschirminhalt), dann sollte Ihr Drucker in der Lage sein, die Grafik, die auf dem Bildschirm zu sehen ist, auszudrucken.

Software

Turbo Prolog läuft unter DOS, dem Standard-Betriebssystem für PCs und ATs. Es ist gleichgültig, ob Sie PC-DOS, MS-DOS oder irgendein MS-DOS-Derivat (wie z. B. Z-DOS) verwenden, solange Sie eine Version von 2.0 ab aufwärts besitzen. Sie sollten mit den Grundfunktionen von DOS vertraut sein: Starten des Systems, Formatieren von Disketten und – bei Festplatten – Verwendung von Unterverzeichnissen.

Natürlich benötigen Sie auf jeden Fall die Turbo Prolog-Disketten. Sie haben drei Originaldisketten erhalten. Eine davon ist die Systemdiskette von Turbo Prolog, die beiden anderen enthalten Beispielprogramme.

Legen Sie sich außerdem ein paar leere Disketten zurecht. Drei benötigen Sie auf jeden Fall, um Kopien von den Originaldisketten anzufertigen. Wenn Sie nicht über eine Festplatte verfügen, brauchen Sie zusätzlich zwei oder drei Arbeitsdisketten.

Der Aufkleber der Programmdiskette weist eine Seriennummer auf. Diese Nummer kennzeichnet Ihr persönliches Exemplar von Turbo Prolog, und deshalb sollten Sie sie irgendwo registrieren. Außerden sollten Sie die Nummer beim Ausfüllen der Garantiekarte eintragen und diese an Borland International schicken. Dadurch werden Sie in das Verzeichnis der Turbo Prolog-Anwender aufgenommen und über Verbesserungen und Änderungen informiert. Beispielsweise erhalten Anwender, die die erste Version von Turbo Prolog gekauft haben und ihre Garantiekarte einschicken, automatisch die Version 1.1 des Programms kostenlos.

Anfertigen von Kopien

Bevor Sie Prolog das erstemal verwenden, sollten Sie von jeder Diskette eine Kopie anfertigen. Keine der Disketten ist kopiergeschützt.

Anmerkung: Vor dem Erstellen von Kopien sollten Sie sich vergewissern, daß die Originaldisketten schreibgeschützt sind. Überprüfen Sie die kleine Schreibschutzkerbe an der rechten Seite der Diskette, und versehen Sie sie wenn nötig mit einem Schreibschutz-Aufkleber. Das Abkleben dieser Kerbe schützt die Diskette davor, beim Kopiervorgang versehentlich beschrieben zu werden.

Legen Sie nun eine DOS-Systemdiskette in Laufwerk A, und schalten Sie Ihren Rechner ein. Falls gefordert, geben Sie das Datum und die Uhrzeit ein. Um

den Kopiervorgang einzuleiten, schreiben Sie das folgende Kommando:

```
A>DISKCOPY A: B:↵
```

Nach Aufforderung legen Sie die Original-Programmdiskette in Laufwerk A. Wenn Ihnen zwei Diskettenlaufwerke zur Verfügung stehen, können Sie in Laufwerk B eine leere Diskette einlegen. Bei der Verwendung nur eines Laufwerks werden Sie zu gegebener Zeit zum Diskettenwechsel aufgefordert.

Nach Abschluß des Kopiervorgangs können Sie die Frage nach einer weiteren Kopie mit *y* beantworten. Legen Sie nun die Beispieldiskette in Laufwerk A ein, und verfahren Sie mit dieser und der zweiten Beispieldiskette genauso, wie bisher beschrieben.

Sobald die drei Disketten kopiert sind, sollten Sie auch die Schreibschutzkerben der neu angelegten Disketten mit Aufklebern versehen, um diese ebenso, wie Sie es bei den Originaldisketten gemacht haben, vor unbeabsichtigtem Schreiben zu schützen. Beschriften Sie die Kopien deutlich, wobei auch die Seriennummer auf der Programmdiskette eingetragen werden sollte. Die Originaldisketten sollten an einem sicheren Ort verwahrt und in Zukunft nur noch die Kopien verwendet wrden.

Drucken der Datei README

Als nächstes sollte die Datei README, die sich auf der Programmdiskette befindet, ausgedruckt werden. Diese enthält Informationen über Turbo Prolog, die nicht im Handbuch zu finden sind. Diese Datei wird mittels des Programms README.COM ausgegeben.

Um diese Datei zu lesen, legen Sie die Programmdiskette in Laufwerk A ein und rufen README auf:

```
A>README↵
```

Auf dem Bildschirm, den Sie so eingestellt haben sollten, daß das Editor-Fenster die gesamte Breite einnimmt, erscheint die erste Seite der Datei.

Es ist empfehlenswert, eine Kopie dieser Datei auszudrucken. Bei älteren Versionen konnte die Datei durch Drücken der Funktionstaste <F2> vom Programm README aus gedruckt werden. Ab Version 1.1 müssen Sie das Programm verlassen und die Datei durch Drücken der Tastenkombination <Ctrl> und <PrtSc> drucken. Drücken Sie also zunächst <Ctrl> <PrtSc>. Geben Sie

dann das Kommando

```
A>TYPE README↵
```

ein, und drücken Sie dann erneut <Ctrl> und <PrtSc>. Dabei halten Sie die <Ctrl>-Taste gedrückt.

Erstellen einer Arbeitskopie

Der nächste Schritt ist das Erstellen einer Kopie von Turbo Prolog, die beim Programmieren verwendet wird. Eine solche Kopie wird auch *Arbeitskopie* genannt. Wenn möglich, sollten Sie diese Kopie auf der Festplatte installieren.

Arbeiten mit der Festplatte

Auf der Festplatte sollten Sie ein neues Unterverzeichnis anlegen, und die Turbo Prolog-Disketten in dieses neue Verzeichnis hineinkopieren. Wenn Sie damit noch keine Erfahrungen gesammelt haben, sollten Sie dies zunächst in einem guten Buch über DOS nachlesen und sich mit dem Erstellen und der Verwendung von Unterverzeichnissen vertraut machen.

Im allgemeinen werden beim Anlegen eines Unterverzeichnisses das Kommando *MD* zum Erstellen, das Kommando *CD* zum Umschalten in dieses Verzeichnis und das Kommando *COPY* zum Kopieren des Disketteninhalts (in Laufwerk A) in dieses neue Verzeichnis verwendet.

`C>CD\↵`	Umschalten ins Hauptverzeichnis
`C>MD PROLOG ↵`	Anlegen eines PROLOG-Verzeichnisses
`C>CD \PROLOG ↵`	Umschalten in das neue Verzeichnis
`C>COPY A:*.* C: ↵`	Kopieren der Diskette

Ab Version 1.1 werden auf den Beispieldisketten auch mehrere Unterverzeichnisse geführt. Sie sollten überlegen, welche Sie davon auf die Festplatte übernehmen wollen, und nur die Dateien aus jedem Verzeichnis kopieren, die Sie brauchen.

Arbeiten mit Diskettenlaufwerken

Wenn Sie keine Festplatte zur Verfügung haben, sollten Sie sich eine Arbeitsdiskette einrichten. Im Grunde genommen benötigen Sie beim Arbeiten mit

Turbo Prolog nur drei Dateien auf der Diskette: PROLOG.EXE, PROLOG.OVL und PROLOG.SYS. Die ersten beiden Dateien enthalten das Prolog-Programm. Bei der dritten handelt es sich um eine Konfigurationsdatei, in der festgelegt wird, wie Turbo Prolog konfiguriert sein muß (siehe Kapitel 2).

Außerden sollten Sie überlegen, die Datei PROLOG.ERR mit auf die Diskette nehmen, da sich in dieser Datei die Texte für die Fehlermeldungen befinden, was in vielen Fällen eine sinnvolle Hilfe sein kann. Ohne diese Datei wird jeweils nur eine Fehlernummer angezeigt, und Sie müßten die Bedeutung dieser Nummer im Handbuch nachschlagen. Weiter gibt es noch die Datei PROLOG.HLP, die Ihre Arbeit durch Hilfsmeldungen unterstützen kann. Diese fünf Dateien dürften ausreichen, um Programme in Turbo Prolog zu erstellen, zu verändern und auszuführen.

Anmerkung: Mit einer derart angelegten Arbeitsdiskette können allerdings nur Compilierungen für internen Gebrauch ausgeführt werden.

Um eine solche Arbeitsdiskette einzurichten, legen Sie die DOS-Diskette in Laufwerk A ein und formatieren eine leere Diskette in Laufwerk B unter Verwendung der Option /s (die Option /s bewirkt, daß Ihre Arbeitsdiskette selbst booten kann):

```
A>FORMAT B:/s ↵
```

Wenn die Formatierung abgeschlossen ist, entfernen Sie die DOS-Diskette und legen die Kopie der Programmdiskette in Laufwerk A ein. Tippen Sie nun:

```
A>COPY PROLOG.* B:/v ↵
```

Damit wird der Kopiervorgang ausgelöst. Nach Abschluß dieser Operation ist Ihre Arbeitsdiskette einsatzbereit. Beschriften Sie diese neue Diskette als *Turbo Prolog-Arbeitsdiskette*, und verwenden Sie sie für die Beispiele in diesem Buch.

Natürlich können Sie auf die soeben beschriebene Weise mehrere Arbeitsdisketten anfertigen. Diese Arbeitsdisketten brauchen nicht mit einem Schreibschutz versehen zu werden, da auf sie unter Umständen auch geschrieben werden soll.

Die wichtigsten Dateien von Turbo Prolog

In der Tabelle A.1 werden die wichtigsten Dateien der Turbo Prolog-Disketten aufgeführt, und ihr Inhalt wird kurz erläutert.

Datei	Beschreibung
PROLOG.EXE	enthält das Turbo Prolog-Programm.
PROLOG.OVL	ist eine Erweiterung zum Prolog-Programm.
PROLOG.SYS	enthält die mit dem Kommando Setup eingestellten Fenstergrößen und -positionen und das Standard-Verzeichnis.
README.COM	ist ein Programm zum Lesen und Drucken der Datei README.
README	ist eine Textdatei mit den neuesten Informationen und Hinweisen für den Anwender.
PROLOG.ERR	enthält die Texte zu den Fehlermeldungen.
PROLOG.HLP	ist eine Datei mit Hilfsinformationen, die über die Taste <F1> erreicht werden können; sie kann mit jedem Editor verändert werden.
PROLOG.LIB	ist die Bibliotheksdatei, die benötigt wird, wenn Dateien für externen Gebrauch compiliert werden.
INIT.OBJ	ist eine Objektdatei, die beim Compilieren für externen Gebrauch zum Linken gebraucht wird.
*.PRO	ist die allgemeine Form mit der Erweiterung PRO; alle so bezeichneten Dateien sind Prolog-Programmdateien und können mit Turbo Prolog oder einem externen Editor bearbeitet werden.

Tabelle A.1: Die wichtigsten Dateien in Turbo Prolog

Anhang B

Verzeichnis der Standard-Prädikate

Im folgenden geben wir ein alphabetisches Verzeichnis der eingebauten oder auch Standard-Prädikate in Turbo Prolog. Jede Beschreibung umfaßt

1. Prädikat-Name und -format
2. Typangaben für jedes Argument
3. Fließmuster für jedes Argument (i - input, also Eingabe oder gebunden, und o – output, also Ausgabe oder frei)
4. Kurze Funktionsbeschreibung

asserta (<Faktum)> (dbasedom) : (i)

Fügt ein *Faktum* vor andere Fakten in einer dynamischen Datenbank ein.

assertz (<Faktum)> (dbasedom) : (i)

Fügt ein *Faktum* nach anderen Fakten in einer dynamischen Datenbank ein.

attribute(Attribut) (integer) : (i) (o)

Setzt den Attributwert für das aktuelle Fenster oder gibt ihn zurück.

back(Schritt) (integer) : (i)

Bewegt die Schildkröte um die angegebene Schrittzahl zurück.

beep

Erzeugt einen Piepston.

bios(Unterbrechungsnummer,RegisterEin,RegisterAus) (integer,regdom,regdom) : (i,i,o)

Ruft die Unterbrechung mit der angegebenen *Unterbrechungsnummer* auf.

bound(Variable) (<Variable>) : (o)

Ist erfolgreich, wenn *Variable* gebunden ist.

char_int(Zeichen,GanzeZahl) (char,integer) : (i,o) (o,i) (i,i)

Wandelt eine ganze Zahl in ein Zeichen oder ein Zeichen in eine ganze Zahl um oder prüft auf Gleichheit.

clearwindow

Löscht das aktuelle Fenster.

closefile(SymbolischerDateiname) (file) : (i)

Schließt die Datei mit dem symbolischen Dateinamen.

comline(Zeile) (string) : (o)

Liest die Parameter der Kommandozeile.

concat(Zeichenkette1,Zeichenkette2,Ergebniszeichenkette)
(string,string,string) : (i,i,o) (o,i,i) (i,o,i) (i,i,i)

Verkettet *Zeichenkette* und *Zeichenkette2* zur *Ergebniszeichenkette*.

consult(DOSDateiname) (string) : (i)

Fügt in die aktuelle Datenbank eine Textdatei ein.

cursor(Zeile,Spalte) (integer,integer) : (i,i) (o,o)

Bewegt den Cursor zu der durch *Zeile* und *Spalte* angegebenen Position oder liefert die aktuelle Cursorposition.

cursorform(Startzeile,Endzeile) (integer,integer) : (i,i)

Setzt die Höhe und vertikale Position des Cursors fest.

date(Jahr,Monat,Tag) (integer,integer,integer) : (i,i,i) (o,o,o)

Liest oder setzt das Datum.

deletefile(DOSDateiname) (string) : (i)

Löscht die angegebene DOS-Datei aus dem aktuellen Verzeichnis.

dir(Pfadname,Dateispezifikation,DOSDateiname) (string,string,string) (i,i,o)

Gibt das aktuelle Verzeichnis aus.

disk(DOSPfad) (string) : (i) (o)

Stellt Standard-Pfad und -Laufwerk ein oder gibt sie aus.

display(Zeichenkette) (string) : (i)

Gibt die *Zeichenkette* im aktuellen Fenster aus. Veränderungen können nicht ausgeführt werden.

dot(Zeile,Spalte,Farbe) (integer,integer,integer) : (i,i,i) (i,i,o)

Setzt an der durch *Zeile* und *Spalte* angegebenen Position einen Punkt in der *Farbe* oder liefert den Farbwert an dieser Stelle. Der Bildschirm muß sich im Grafikmodus befinden.

edit(Eingabezeichenkette,Ausgabezeichenkette) (string,string) : (i,o)

Gibt die *Eingabezeichenkette* aus und erzeugt nach möglicher Veränderung die *Ausgabezeichenkette*.

editmsg(Eingabezeichenkette,Ausgabezeichenkette,Titellinks,Titelrechts,Meldung,Position,Hilfsdateiname,Code)(string,string,string,string,string,integer, string,integer) : (i,o,i,i,i,i,i,o)

Gibt die *Eingabezeichenkette* aus und erzeugt nach möglicher Veränderung die *Ausgabezeichenkette*. Fügt Text links oder rechts in die Kopfzeile ein, gibt eine Meldung am unteren Bildschirmrand aus, benennt die Datei mit den Hilfsinformationen und den Code, mit dem das Editieren abgeschlossen wurde.

eof(SymbolischerDateiname) (file) : (i)

Prüft an der aktuellen Position die Datei mit dem symbolischen Dateinamen auf Endekennzeichen.

existfile(DOSDateiname) (string) : (i)

Ist erfolgreich, wenn der DOS-Dateiname im aktuellen Verzeichnis vorhanden ist.

exit

Beendet den Programmlauf und führt ins Turbo Prolog-Menü zurück.

fail

Erzwingt das Scheitern der aktuellen Regel.

field_attr(Zeile,Spalte,Länge,Attribut)(integer,integer,integer,integer): (i,i,i,i) (i,i,i,o)

Setzt oder gibt das aktuelle Feldattribut zurück.

field_str(Zeile,Spalte,Länge,Zeichenkette) (integer,integer,integer,string) : (i,i,i,i) (i,i,i,o)

Schreibt die *Zeichenkette* in das durch *Zeile, Spalte* und *Länge* gekennzeichnete Feld oder liest sie.

filemode(SymbolischerDateiname,Dateimodus) (file,integer) : (i,i) (i,o)

Ermöglicht Dateizugriff im Binärmodus.

filepos(SymbolischerDateiname,Dateiposition,Modus) (file,real,integer) (i,i,i) (i,o,i)

Setzt den Dateizeiger für die nächste Schreiboperation.

file_str(DOSDateiname,Zeichenkettenvariable) (string,string) : (i,o) (i,i)

Liest Zeichen aus der angegebenen Datei.

findall(Variable,<Atom>,Variablenliste)

Sammelt die Werte für das Backtracking in einer Liste. Die Liste muß im Abschnitt *domains* als solche definiert werden.

flush(SymbolischerDateiname) (file) : (i)

Entleert den Pufferinhalt in die aktuelle Datei.

forward(Schritt) (integer) : (i)

Bewegt die Schildkröte um die angegebenen Schritte vorwärts.

free(Variable) (<Variable>) : (o)

Ist erfolgreich, wenn die *Variable* frei ist.

frontchar(Zeichenkette,ErstesZeichen,Restzeichenkette)(string,char,string) (i,o,o) (i,i,o)

Liefert das erste Zeichen einer Zeichenkette oder testet es.

frontstr(AnzahlZeichen,Eingabezeichenkette,Ergebniszeichenkette,Restzeichenkette) (integer,string,string,string) : (i,i,o,o)

Entnimmt der *Eingabezeichenkette* vorne die angegebene Anzahl von Zeichen.

fronttoken(Eingabezeichenkette,Token,Restzeichenkette) (string,string,string) : (i,o,o) (i,i,o) (i,o,i) (i,i,i) (o,i,i)

Entnimmt der *Eingabezeichenkette* das erste Token (Wort) oder testet es.

gotowindow(Fensternummer) (integer) : (i)

Schaltet in das Fenster mit der angegebenen Nummer um.

graphics(Modus,Palette,Hintergrund) (integer,integer,integer) : (i,i,i)

Versetzt den Bildschirm in den Grafikmodus.

inkey(Zeichen) (char) : (o)

Liest ein einzelnes Zeichen vom Standard-Eingabegerät; scheitert, wenn zur Zeit der Ausführung kein Zeichen eingegeben wird.

isname(Zeichenkette) (string) : (i)

Ist erfolgreich, wenn die *Zeichenkette* ein gültiger Name in Turbo Prolog ist.

keypressed

Ist erfolgreich, wenn eine Taste gedrückt wurde; andernfalls scheitert es.

left(Winkel) (integer) : (i) (o)

Dreht die Schildkröte um den angegebenen *Winkel* gegen den Uhrzeigersinn.

line(Zeile1,Spalte1,Zeile2,Spalte2,Farbe) (integer,integer,integer,integer,integer) : (i,i,i,i,i)

Zeichnet im Grafikmodus eine Linie.

makewindow(Fensternummer,Bildschirmattribut,Rahmenattribut,Titel,Zeile,Spalte,Höhe,Breite) (integer,integer,integer,string,integer,integer,integer,integer) : (i,i,i,i,i,i,i,i) (o,o,o,o,o,o,o,o)

Definiert einen Bildschirmbereich als Fenster oder gibt die aktuellen Fensterparameter zurück.

membyte(Segment,Offset,Byte) (integer,integer,integer) : (i,i,i) (i,i,o)

Speichert oder liest ein Speicherbyte an der angegebenen Adresse.

memword(Segment,Offset,Word) (integer,integer,integer) : (i,i,i) (i,i,o)

Speichert oder liest ein Speicherwort an der angegebenen Adresse.

nl

Bewirkt einen Zeilenvorschub.

not(<Atom>)

Ist erfolgreich, falls *Atom* ein Ziel darstellt, das bei der Ausführung scheitert.

openappend(SymbolischerDateiname,DOSDateiname) (file,string) : (i,i)

Eröffnet eine Datei zum Anfügen von Daten.

openmodify(SymbolischerDateiname,DOSDateiname) (file,string) : (i,i)

Eröffnet eine Datei zum Ändern von Daten.

openread(SymbolischerDateiname,DOSDateiname) (file,string) : (i,i)

Eröffnet eine Datei zum Lesen von Daten.

openwrite(SymbolischerDateiname,DOSDateiname) (file,string) : (i,i)

Eröffnet eine Datei zum Schreiben von Daten.

pencolor(Farbe) (integer) : (i)

Legt die Schreibfarbe der Schildkröte fest.

pendown

Senkt den Stift der Schildkröte zum Zeichnen.

penpos(Zeile,Spalte,Richtung) (integer,integer,integer) : (i,i,i) (o,o,o)

Setzt oder liest die Position der Schildkröte.

penup

Hebt den Stift der Schildkröte.

portbyte(Portnummer,Wert) (integer,integer) : (i,i) (i,o)

Liest ein Byte vom Port mit der angegebenen Nummer oder sendet eins dorthin.

ptr_dword(Zeichenkettenvariable,Segment,Offset)(string,integer,integer): (i,o,o) (o,i,i)

Setzt oder liefert die Segment- und Offsetadresse.

readchar(Zeichenvariable) (char) : (o)

Liest ein Zeichen vom aktuellen Eingabegerät.

readdevice(SymbolischerDateiname) (symbol) : (i) (o)

Ordnet dem aktuellen Eingabegerät den symbolischen Dateinamen zu.

readint(GanzzahlVariable) (integer) : (o)

Liest eine ganze Zahl vom aktuellen Eingabegerät.

readln(Zeichenkettenvariable) (string) : (o)

Liest eine Zeichenkette vom aktuellen Eingabegerät.

readreal(DezimalzahlVariable) (real) : (o)

Liest eine Dezimalzahl vom aktuellen Eingabegerät.

readterm(Bereich,Term) (<Name>,<Variable>) : (i,o)

Liest ein Objekt, das mit dem Prädikat *write* geschrieben wurde. Ermöglicht den Zugriff auf Fakten in Dateien.

removewindow

Entfernt das aktuelle Fenster vom Bildschirm.

renamefile(AlterDOSDateiname,NeuerDOSDateiname) (string,string) : (i,i)

Benennt eine DOS-Datei um.

retract(<Faktum>) (dbasedom) : (i)

Entfernt das erste Faktum aus der Datenbank.

right(Winkel) (integer) : (i) (o)

Dreht die Schildkröte um den angegebenen *Winkel* mit dem Uhrzeigersinn.

save(DOSDateiname) (string) : (i)

Speichert die dynamische Datenbank auf Diskette

scr_attr(Zeile,Spalte,Attribut) (integer,integer,integer) : (i,i,i) (i,i,o)

Setzt oder liefert die Zeichenattribute an der durch *Zeile* und *Spalte* angegebenen Position.

scr_char(Zeile,Spalte,Zeichen) (integer,integer,char) : (i,i,i) (i,i,o)

Setzt oder liefert das Zeichen an der durch *Zeile* und *Spalte* angegebenen Position.

scroll(AnzahlReihen,AnzahlSpalten) (integer,integer) : (i,i)

Rollt das aktuelle Fenster.

shiftwindow(Fensternummer) (integer) : (i) (o)

Schaltet auf das Fenster mit der angegebenen Nummer um oder bindet die Nummer an das aktuelle Fenster.

sound(Dauer,Frequenz) (integer,integer) : (i,i)

Spielt eine Note in der angegebenen Frequenz mit der spezifizierten Dauer.

storage(StackGröße,HeapGröße,TrailGröße) (real,real,real) : (o,o,o)

Gibt die verfügbaren Speicherbereiche an.

str_char(Zeichenkette,Zeichen) (string,char) : (i,o) (o,i) (i,i)

Macht aus Zeichenketten mit einem Zeichen ein Zeichen und umgekehrt.

str_int(Zeichenkette,Ganzzahl) (string,integer) : (i,o) (o,i) (i,i)

Macht aus Zeichenketten ganze Zahlen oder umgekehrt. Dabei entspricht die ganze Zahl dem ASCII-Wert.

str_len(Zeichenkette,Länge) (string,integer) : (i,i) (i,o)

Liefert die Länge einer Zeichenkette oder testet sie.

str_real(Zeichenkette,Dezimalzahl) (string,real) : (i,o) (o,i) (i,i)

Macht aus einer Zeichenkette eine Dezimalzahl oder umgekehrt.

system(DOSKommandoZeichenkette) (string) : (i)

Führt von Turbo Prolog aus ein DOS-Kommando aus.

text

Schaltet den Bildschirm in den Textmodus zurück.

time(Stunden,Minuten,Sekunden,Hundertstelsekunden)(integer,integer,integer,integer) : (i,i,i,i) (o,o,o,o)

Liest oder stellt die Systemzeit ein.

trace(Zustand) (symbol) : (i) (o)

Schaltet die Programmablaufverfolgung ein oder aus.

upper_lower(Großbuchstabenzeichenkette, Kleinbuchstabenzeichenkette) (string,string) : (i,i) (i,o) (o,i)

Wandelt Großbuchstaben in Kleinbuchstaben um oder umgekehrt oder prüft auf Äquivalenz.

window_attr(Attribut) (integer) : (i)

Weist dem aktuellen Fenster einen Attributwert zu.

window_str(Bildschirmzeichenkette) (string) : (i) (o)

Liest oder schreibt den aktuellen Fensterinhalt.

write(E1,E2,E3,...,En) ((i)*)

Schreibt auf das aktuelle Ausgabegerät.

writedevice(SymbolischerDateiname) (symbol) : (i) (o)

Weist dem aktuellen Ausgabegerät den symbolischen Dateinamen zu oder bindet den symbolischen Dateinamen an das aktuelle Ausgabegerät.

writef(Formatzeichenkette,Arg1,Arg2,Arg3,...) (i,(i)*)

Gibt eine formatierte Ausgabe auf dem aktuellen Ausgabegerät aus.

Anhang C

Reservierte Wörter

Die folgenden Wörter sollten in Turbo Prolog nicht als Relations- oder Objektnamen verwendet werden.

abs
and
arctan
asserta
attribute
back
beep
bios
bitand
bitleft
bitnot
bitor
bitright
bitxor
bound
char_int
check_determ
clauses
clearwindow
closefile
code
comline
concat
config
consult
cos
cursor
cursorform
database
date
deletefile
diagnostics
dir
disk
display
div
domains
dot
edit
editmsg
eof
existfile
exit
exp
fail
field_attr
filemode
file_str
filepos
findall
flush
forward
free
frontchar
frontstr
fronttoken
global
goal
gotowindow
if
include
inkey
isname
keypressed
left
length
line
ln
log
makewindow
membyte
memword
mod
nl
nobreak
not
nowarnings
openappend
openmodify
openread
openwrite
or
pencolor
pendown
penpos
penup
port_byte
predicates
project
ptr_dword
random
readchar
readdevice
readint
readln
readreal
reference
removewindow
renamefile
retract
right
round
save
scr_attr
scr_char
scroll
shiftwindow
shorttrace
sin
sound
sqrt
storage
str_char
str_int
str_len
str_real
system
tan
text
time
trace
trail
upper_lower
window_attr
window_str
write
writedevice
writef

Anhang D

Glossar

Algorithmus — Eindeutiges Verfahren zur Lösung einer Klasse von Problemen, das in endlicher Zeit eine Lösung liefert (siehe Heuristik).

Argumente — Parameter eines Prädikats.

Atom — Begriff aus dem Standard-Prolog von Clocksin und Mellish. Gemeint ist ein Prädikat mit Objekten und Variablen.

Attribute — Eigenschaften oder Merkmale eines Objekts.

Backtracking — Ein Prolog-Prozeß, bei dem das Programm bis zu einem Punkt zurückgeht, von wo aus es einen neuen Lösungsweg mit anderen Variablenwerten sucht.

Befragungs-Paradigma — Anwenderschnittstelle, bei der der Anwender mit Hilfe von interaktiven Fragen durch den Problemraum zum Ziel geführt wird (siehe Paradigma).

Bereich (Domain) — Wertetypen, die ein Argument in einem bestimmten Prädikat-Ausdruck annehmen kann, oder ein definierbarer Wissensbereich über ein bestimmtes Thema.

Binden — Zuweisung eines Wertes an eine Variable.

Breitensuchverfahren — Suchverfahren, bei der zunächst alle Regeln auf der gleichen Ebene überprüft werden, bevor die nächste Ebene betreten wird (siehe Tiefensuchverfahren).

Break — Verbindung zwischen Kopf und Rumpf einer Regel; wird durch das Symbol :- dargestellt.

Compiler — Übersetzer, der das Quellprogramm in eine direkt ausführbare Form umwandelt.

Cut — Siehe Schnitt.

Datenbank — Sammlung von bekannten Fakten und Regeln in einem bestimmten Bereich (siehe dynamische Datenbank; statische Datenbank).

Deklarative Sprache — Sprache, in der das Wissen über einen Bereich mit Hilfe von symbolischen Ausdrucksweisen in einer objektiven Form dargestellt wird. Probleme werden durch Vorgabe eines spezifizierten Ziels gelöst, für das der Computer einen Wert bestimmen muß. Die Sprache ist nicht-prozedural (d. h. sie verwendet keinen Algorithmus), aber Heuristiken und anderes Wissen über die Prozedur können angegeben werden, und der Computer kann auf diese zurückgreifen, wenn er versucht, einen Lösungsweg zu dem angegebenen Ziel zu finden. Prolog und LISP sind auch deklarative Sprachen.

Diagnose-System — Typ von Expertensystem, bei dem Symptome zu Ursachen in Beziehung gesetzt werden.

Domain — Siehe Bereich.

Dynamische Datenbank — Teil der Datenbank, in dem während der Programmausführung Fakten hinzugefügt oder aus dem Fakten entfernt werden können. Das Hinzufügen geschieht mit den Prädikaten *asserta* oder *assertz* und das Entfernen mit dem Prädikat retract (siehe Datenbank; statische Datenbank).

Eingebaute Prädikate — Siehe Standard-Prädikate.

Entscheidungsbaum
Struktur für den Entscheidungsprozeß eines Expertensystems, bei der die Fragen als Knoten und die möglichen Antworten als Verbindungen gezeichnet werden.

Expertensystem
Intelligentes Computer-Programm, das Wissen und Schlußfolgerungs-Mechanismen verwendet, um Probleme zu lösen, die so schwierig sind, daß ein beträchtliches Maß an Fachwissen erforderlich ist.

Faktum
Aufstellen einer Behauptung über ein Objekt oder eine Beziehung zwischen Objekten, wie z. B.:

```
ist_ein(tiger,säugetier).
```

Ein Faktum endet immer mit einem Punkt. Der Abschnitt clauses eines Prolog-Programms besteht aus Fakten und Regeln.

Formale Schlußfolgerungen
Prozeß, bei dem ein Problem durch Inferenz oder IF-THEN-Regeln gelöst wird.

Funktor
Name für ein zusammengesetztes Objekt.

Goal
siehe Ziel.

Heuristik
Verfahren, das den Problemraum auf der Suche nach einer speziellen Problemlösung einengt. Heuristische Verfahren garantieren nicht unbedingt eine Lösung und (wenn eine Lösung existiert) auch nicht immer den effektivsten Lösungsweg (siehe Algorithmus).

Hierarchie
Struktur, in der einige Objekte anderen untergeordnet sind. Objekte der oberen Ebene werden auch *Eltern* und Objekte der unmittelbar darunterliegenden Ebene *Kinder* genannt.

Inferenz
Schlußfolgerungs-Prozeß, bei dem neue Fakten aufgrund bekannter Fakten mit Hilfe von Regeln und des *Modus ponens* abgeleitet werden.

Inferenz-Maschine
Teil eines Expertensystems, das unter Verwendung von Regeln einer Datenbank neue Fakten aus bereits vorhandenen Fakten ableitet. Die Inferenz-Maschine steuert den Ablauf und die Reihenfolge der Schlußfolgerungen.

Infix-Notation
Mathematische Schreibweise, bei der der Operator in einem mathematischen Ausdruck zwischen den Operanden oder Werten steht, auf die er sich bezieht (siehe Präfix-Notation; Suffix-Notation).

Interpreter
Interpretierer, der den Quellcode eines Programms Schritt für Schritt in eine ausführbare Form umwandelt und auch sogleich ausführt (siehe Compiler).

Klausel
Faktum oder Regel, endet mit einem Punkt.

Konfliktlösung
Entscheidungsprozeß in einem Produktionssystem, welche Regel gewählt werden soll, wenn zwei oder mehr Regeln mit der Zielaussage unifizieren.

Kopf
Erstes Element in einer Liste oder Behauptung einer Regel (Regelkopf).

Künstliche Intelligenz
Teilgebiet der Informatik, das sich mit Konzepten und Methoden symbolischer Schlußfolgerungen durch einen Computer und der symbolischen Darstellung von Wissen befaßt.

Dabei wird als Ziel verfolgt, daß der Computer Verhaltensweisen in einer Weise aufzeigt, die allgemein als "intelligent" angesehen wird.

Liste Objekt, das eine geordnete Gruppe von Elementen enthält.

Match Vergleichen von Funktoren, Stelligkeiten und Argumenten eines Ziels mit Fakten und Regelköpfen, um zu einer Übereinstimmung zu kommen (siehe Unifikation).

Modus ponens Regel aus der formalen Logik, die besagt, daß, wenn B aus A folgt und A wahr ist, dann B auch wahr ist.

NIL Leere Liste, wird durch [] dargestellt.

Objekt In Turbo Prolog konstantes Argument eines Prädikats, sonst eine physikalische oder gedankliche Einheit, die ein oder mehrere Attribute haben kann.

Objekt-orientierte Sprache Deklarative Sprache, bei der Wissen über einen Bereich in Objekten ausgedrückt wird. Jedes Objekt hat ein oder mehrere Attribute, und diese besitzen Werte.

Paradigma Spezielle Problemlösungsmethode für bestimmte Anwendungen; das diagnostisch-beschreibende Befragungs-Paradigma ist z.B. ein in Prolog übliches Paradigma.

Parser Syntax-Analysierer, der einen Ausdruck in seine Komponenten zerlegt und die Syntax bestimmt (siehe Syntax).

Prädikat Funktion, die den Wert wahr oder falsch hat und eine Eigenschaft oder Beziehung ausdrückt.

Präfix-Notation Mathematische Schreibweise, bei der der Operator in einem mathematischen Ausdruck vor den Operanden oder Werten steht, auf die er sich bezieht (siehe Infix-Notation; Suffix-Notation).

Problemraum Netzwerk aus Knoten und Verbindungen zur Lösung eines Problems in einem festgelegten Bereich. Die Knoten stellen Ziele und Subziele dar. Die Verbindungen symbolisieren Operationen oder Pfade zwischen den Zielen und Subzielen. Der Problemraum wird auch Suchbaum genannt.

Produktion Eine Regel; die einem regelbasierenden System zugrundeliegende Speichereinheit.

Produktionssystem Siehe Regelbasierendes System.

Prototyp Vorabversion eines Systems. Bei einem Expertensystem repräsentiert ein Prototyp einen wesentlich kleineren Bereich als das endgültige System.

Prozedurale Sprache Sprache, in der zur Lösung eines Problems ein Algorithmus in einer objektiven Form angegeben werden kann, der für zahlreiche Probleme des gleichen Typs wiederholt zum Einsatz kommen kann. Prozedurale Sprachen sind z. B. BASIC, Pascal, C und COBOL.

Rahmen Zugrundeliegende Speichereinheit bei einem rahmenbasierenden System (siehe Rahmenbasierendes System).

Rahmenbasierendes System Eine Art der Wissensdarstellung, bei der ein Objekt durch einen Rahmen dargestellt wird und die Attribute des Objekts durch Slots (Schubfächer) in diesem Rahmen, die die Attributwerte enthalten.

Regel Klausel, die eine Beziehung zwischen Fakten definiert. Der erste Teil einer Regel heißt *Kopf* oder *Behauptung*, der zweite Teil heißt *Rumpf*. Beide Teile werden durch den Operator IF miteinander verbunden.

Regelbasis Teil eines Expertensystems, in dem statische Informationen über einen bestimmten Bereich gespeichert werden (während eines Programmlaufs wird daran nichts verändert). In Prolog ist die statische Datenbank die Regelbasis (siehe statische Datenbank).

Regelbasierendes System Art der Wissensdarstellung, in der Wissen über ein bestimmtes Fachgebiet in Regeln ausgedrückt wird, die Beziehungen zwischen den Fakten aufzeigen.

Rekursion Methode, bei der ein zu definierender Begriff zu seiner eigenen Definition gebraucht wird.

Relation Name, der die Eigenschaften einer Menge von Objekten (oder Objekten und Variablen) und ihre Beziehungen untereinander beschreibt, sozusagen der Funktor-Teil eines Prädikats.

Restliste Rest einer Liste ohne Kopf. Die Restliste einer einelementigen Liste ist NIL oder die leere Liste, die durch zwei eckige Klammern [ ] dargestellt wird.

Rückwärtsverkettung Strategie, bei der, ausgehend von einer Zielaussage, zurückgegangen wird, um diese zu beweisen (siehe Vorwärtsverkettung).

Schnitt Prolog-Operator, der immer erfolgreich ist und Prolog auf die Lösungen und Variablenwerte beschränkt, die bis zu diesem Punkt gefunden wurden. Ein Schnitt verhindert Backtracking über die Stelle, an der sich der Schnitt befindet, hinaus.

Semantik Bedeutung oder Interpretation eines Ausdrucks (siehe Syntax).

Slot Speichereinheit eines rahmenbasierenden Systems. Slots (Schubladen) können den

Namen eines Objekts, seine Eigenschaften (Attribute), Werte, Standardwerte, Grenzen für andere Werte, Prozeduren und Regeln zum Ermitteln von Werten, Zeiger auf andere Rahmen und andere Objekt-bezogene Informationen enthalten.

Standard-Prädikate — Prolog-Prädikate, die fester Bestandteil einer bestimmten Implementation sind.

Statische Datenbank — Teil der Datenbank, der während einer Programmausführung unverändert bleibt (siehe Datenbank; dynamische Datenbank).

Status — Jeder Knoten in einem Problemraum.

Subziel — Ziel, das Teil eines anderen Ziels ist.

Suffix-Notation — Mathematische Schreibweise, bei der der Operator in einem mathematischen Ausdruck hinter den Operanden oder Werten steht, auf die er sich bezieht (siehe Infix-Notation; Präfix-Notation).

Stelligkeit — Anzahl der Parameter in einem Prädikat.

Suchbaum — Siehe Problemraum.

Syntax — Regeln zur Festlegung der Reihenfolge von Symbolen in einem Ausdruck. Das Überprüfen der Syntax ist eine Stärke von Computern, während die Interpretation der Semantik eine Schwäche darstellt (siehe Semantik).

Term — Objekt, Liste oder zusammengesetztes Objekt in Turbo Prolog.

Tiefensuchverfahren — Suchverfahren, bei dem jeder mögliche Lösungsweg zu einem angegebenen Ziel so tief wie möglich verfolgt wird, bis das Ziel bewiesen oder widerlegt werden kann (siehe Breitensuchverfahren).

Unifikation Prozeß, bei dem Prolog eine Übereinstimmung eines Ziels oder Subziels mit den Fakten oder Köpfen von anderen Regeln sucht.

Variable Name, der Platzhalterfunktion für einen Wert hat.

Vorwärtsverkettung Strategie, bei der, ausgehend von bekannten Fakten, vorwärtsgegangen wird, um ein Ziel für diese bekannten Fakten zu finden (siehe Rückwärtsverkettung).

Ziel (Goal) Eine oder mehrere Relationen mit Objekten oder Variablen, die vom Prolog-Programm bewiesen werden sollen.

Zusammengesetztes Objekt Objekt, das ein anderes Objekt in einer Art hierarchischer Struktur enthält. Ein zusammengesetztes Objekt besteht aus einem Funktor und einer Liste von untergeordneten Objekten, die in Klammern eingeschlossen und durch Kommas voneinander getrennt sind.

Zusammengesetztes Ziel Ziel, das mindestens zwei Subziele enthält.

Anhang E

Literaturverzeichnis

Barr, A./Cohen, P. R./Feigenbaum, E. A.: The Handbook of Artificial Intelligence, Vol I, II und III. William Kaufmann, Los Altos 1981, 1982

Bundy, A.: Artificial Intelligence. University Press 1980

Bundy, A.: The Computer Modelling of Mathematical Reasoning. Academic Press 1983

Clark, K. L./McCabe, F. G.: micro-PROLOG: Programming in Logic. Prentice/Hall 1984.

Clocksin, W. F./Mellish, C. C.: Programming in Prolog. Springer 1981

Covington, M.: Programming in Logic. PC Technical Journal 3, no. 12 (December 1985): 82-95

Cuadrado, C. Y./Cuadrado, J.: Prolog Goes to Work. Byte 10, no. 8 (August): 151-158

Ennals, J. R.: Beginning micro-PROLOG. Ellis Horwood Heinemann 1983

Forsyth, R.: Expert Systems: Principles and Case Studies. Chapman and Hall, London 1984

Giannesini, F./Kanoui, H./Pasero, R./van Caneghem, M.: PROLOG. Addison Wesley Verlag (Deutschland) 1986

Godenberg, J.: Experts on Call. PC World 3, no. 9 (September 1985): 192-199

Habel, Ch. (Hrsg.): Künstliche Intelligenz. Informatik-Fachberichte 93. Springer 1985

Harmon, P./King, D.: Expert Systems: Artificial Intelligence in Business. John Wiley, New York 1985

Hayes-Roth, F./et al: Building Expert Systems. Addison Wesley, Reading, Mass. 1983

Holland, G.: Problemlösen mit micro-PROLOG. B. G. Teubner Stuttgart 1986

Kowalski, R.: Logic for Problem Solving. Elsevier, North-Holland 1979

Lloyd, J. W. Foundations of Logic Programming. Springer 1984

McCabe, F. G./Clark, K. L./Steel, B. D.: micro-PROLOG 3.1, Programmers Reference Manual. Logic Programming Associates Ltd, London 1984

Nilsson, N. J.: Principles of Artificial Intelligence. Springer 1982

Siekmann, H. J. (Hrsg.): Künstliche Intelligenz. Informatik-Fachberichte 59. Springer 1982

Townsend, C./Feucht, D.: Personal Expert Systems: Designing and Programming. Blue Ridge Summet, Penn.: Tab 1985

Winston, H. P.: Artificial Intelligence. Addison-Wesley 1984

Anhang F

Ausgewählte Lösungen

Kapitel 3

1. a. Relation: adresse
   Argumente: hans, berlin

   b. Relation: alter
   Argumente: renate, 14

   c. Relation: computer
   Argument: ibm_pc

2. Gültige Objektnamen sind a und e.

3. a. gültig (string)

   b. nicht gültig

   c. gültig (Objekt)

   d. nicht gültig

   e. nicht gültig

   f. gültig (symbol)

   g. gültig (real)

   h. nicht gültig

Kapitel 4

1. a. ort(X,brd)

   b. ort(florenz,X)

2. a. X=erkältung
   1 Solution

   b. X=erkältung, Y=erkältung
   X=grippe, Y=erkältung
   2 Solutions

   c. X=erkältung, Y=leichte_gliederschmerzen
   X=erkältung, Y=schnupfen
   X=grippe, Y=schüttelfrost
   X=grippe, Y=starke_gliederschmerzen
   X=grippe, Y=schnupfen
   X=grippe, Y=leichter_husten
   6 Solutions

   d. X=schnupfen
   1 Solution

Kapitel 5

1. a. Kopf: annahme(meningitis)
   Rumpf:symptom(fieber),symptom(kopfschmerzen),
   symptom(hautflecken)

   b. Kopf: annahme(regen)
   Rumpf:barometer(fallend),wind(west)

2. a. 1

   b. 2

   c. 1

Kapitel 6

1. Wenn das Programm die Klauseln durchprüft, wird immer wieder die gleiche Frage ausgegeben. Das Programm hat keine Möglichkeit, bereits beantwortete Fragen zu speichern.

Kapitel 7

1.
```
domains
        name,strasse,postleitzahl,stadt = string

predicates
        adresse(name,strasse,postleitzahl,stadt)
        los

clauses
        los:-
        adresse(Name,Strasse,Postleitzahl,Stadt),
        write(Name),nl,
        write(Strasse),nl,nl,
        write(Postleitzahl,"",Stadt),nl,
        fail.

        los.

    adresse("Rainer Fuchs","Torweg 3","5300","Bonn").
    adresse("Thea Kroll","Rheinallee 1","5100","Aachen").
    adresse("Max Liebig","Heerstr. 11","5000","Köln").
    adresse("Rosi Stein","Am Markt 7","8132","Tutzing").
```

2.
```
/* BEISPIEL FÜR DEN AUSSCHLUSS VON OBJEKTEN */

predicates
        ort(string,string)
        los      landpr(string)

clauses
        los:-
        writef("%-10 %5 \n","STADT","LAND"),
        fail.

    los:-
        ort(Stadt,Land),
        not(landpr(Land)),
        writef("%-10 %2 \n",Stadt,Land),
        fail.
```

```
los.

landpr("England").

ort("Tokio","Japan").
ort("London","England").
ort("Amsterdam","Holland").
```

3. Wenn die *los*-Klauseln vertauscht würden, ist die erste *los*-Klausel immer erfolgreich, und die zweite wird nie erreicht.

Kapitel 8

1.
```
domains
    name,kennwort = symbol

    predicates
        eingabe(name,kennwort)
        logon(integer)
        anwender(name,kennwort)
        los

    clauses

    los:-

logon(3),

write("Sie sind jetzt angemeldet"),nl.

    logon(0):-,

    write("Sie sind leider unbefugt."),

    fail.

    logon(_):-
        eingabe(Name,Kennwort),
        anwender(Name,Kennwort).

    logon(X):-
        write("Ungültige Eingabe"),nl,
        XX = X - 1,
        logon(XX).
```

```
eingabe(Name,Kennwort):-
    write("Geben Sie bitte Ihren Namen ein: "),
    readln(Name),nl,
    write("Geben Sie das Kennwort ein: "),
    readln(Kennwort),nl.
    anwender(heiner,zebra).
    anwender(helena,nil).
    anwender(robert,oper).
```

Kapitel 9

1. Das Prädikat *ort* ist nicht-deterministisch. Sie können das feststellen, indem Sie den Compiler-Befehl *check_determ* oder *diagnostics* an den Anfang des Programms stellen und dieses dann compilieren und ausführen. Das Programm funktioniert zwar, aber durch Einfügen des Schnitts kann die Warnmeldung ausgeschaltet werden:

```
ort("Tokio","Japan"):-.
```

Kapitel 10

1.
```
predicates

minimum(integer,integer,integer)

    clauses

minimum(X,Y,X):-

    X <= Y,.


minimum(X,Y,Y):-
        Y < X.
```

2.
```
predicates

    los

tab(integer)

clauses

los:-
```

```
tab(20),
write("Das ist ein Test."),nl.
tab(0):-.
tab(N):-
write(" "),
NN = N - 1,
tab(NN).
```

Kapitel 11

1. a. Funktor: auto
   Komponenten: baujahr, fabrikat, farbe

   b. Funktor: teil
   Komponenten: nummer, beschreibung, bestand, preis

   c. Funktor: ort
   Komponenten: stadt, land

   d. Funktor: computer
   Komponenten: (keine)

2.
```
domains
     element = teil(nummer,beschreibung,bestand,preis)
     nummer,bestand = integer
     beschreibung =  string
     preis = real

predicates
     lieselement(element)
     los

clauses
     los:-
     lieselement(Teil),nl,

     write(Teil),nl.
     lieselement(teil(Nummer,Beschreibung,Bestand,Preis)):-
     write("Teilenummer: "),
     readint(Nummer),
     write("Beschreibung :"),
     readln(Beschreibung),
     write("Bestand: "),
     readint(Bestand),
     write("Preis: "),
     readreal(Preis).
```

Kapitel 12

3.
```
domains

    mitglied = adresse(name,strasse,postleitzahl,stadt)

    name,strasse,postleitzahl,stadt = string

    database

    xadresse(mitglied)

predicates

    liesname(mitglied)

    los

    opendatei

clauses

    los:-

    opendatei,

    liesname(Adresse),nl,
    asserta(xadresse(Adresse)),

    save("post.dat").

    liesname(adresse(Name,Strasse,Postleitzahl,Stadt)):-

    write("Name: "),readln(Name),

    write("Straße: "),readln(Strasse),

    write("PLZ: "),readln(Postleitzahl),

    write("Stadt: "),readln(Stadt).

    opendatei:-

    existfile("post.dat"),

    consult("post.dat").

    opendatei.
```

Kapitel 13

1. a. Kopf: 2
   Rest: [56,1,9,3,41,87]

   b. Kopf: rot
   Rest: [blau,schwarz,orange]

   c. Kopf: albert
   Rest: [franz,lilli,rosi]

   d. Kopf: (undefiniert)
   Rest: (undefiniert)

2. Ändern Sie die ersten Klauseln wie folgt:

```
domains
    hliste = annahme*
    annahme = symbol

predicates
    los
    lauf1(hliste)
    inhalt(hliste)
    diagnose(symbol)

clauses
    los:-
    write("MEDIZINISCHES DIAGNOSE-SYSTEM"),nl,
        inhalt(X),lauf1(X).
    lauf1([]):-
    write("\nIch kann Ihr Problem"),nl,
    write("nicht lösen. \n"),lösch_fakten.
    lauf1([K:]):-
        Y = K,diagnose(Y),!,nl,lösch_fakten.
    lauf1([_:Rest]):-
        !,lauf1(Rest).
    inhalt([
        erkältung,
        grippe,
        masern,
        röteln,
        mumps,
        windpocken,
        keuchhusten]).
```

Kapitel 14

1.
```
predicates
          teilkette(integer,integer,string,string)

clauses
          teilkette(1,Länge,Eingabekette,Ausgabekette):-

frontstr(Länge,Eingabekette,Ausgabekette,_),.

          teilkette(Startpos,Länge,Eingabekette,Ausgabeket-
          te):-

          X = Startpos - 1,
          frontstr(X,Eingabekette,_,Hilfskette),
          frontstr(Länge,Hilfskette,Ausgabekette,_).
```

3.
```
predicates
          zentrier(string)
          leerzeichen(integer)

clauses
          zentrier(Text):-
          str_len(Text,Länge),
          X = (80 - Länge) / 2,
          leerzeichen(X),
          write(Text),nl.
          leerzeichen(0):-.
          leerzeichen(N):-
          write(" "),
          NN = N - 1,
          leerzeichen(NN).
```

Kapitel 15

1. Der Rahmen für ein Fenster beansprucht in allen Modi zwei Spalten und zwei Reihen.

2.
```
predicates
    los
clauses
    los:-
    makewindow(1,7,7,"Test",10,30,8,20),

write("Das ist ein Test"),nl,

readchar(_).
```

Kapitel 16

1.
```
domains
    datei = xausgabe

predicates
    los
    datenladen
    einlesen(char)
    eingabe(char)

clauses
    los:-
    datenladen,
    eingabe(X),
    /* lesen bis zur Eingabe von # */
    einlesen(X),
    closefile(xausgabe),
    writedevice(screen).

datenladen:-
    existfile("aus.dat"),
    openappend(xausgabe,'aus.dat'),.

datenladen:-
    openwrite(xausgabe,'aus.dat').

einlesen("#"):-.
einlesen("\13"):-
    ! ,
    writedevice(xausgabe),
    write('"13\10",
    writedevice(screen),nl,
    eingabe(X),
    einlesen(X).

einlesen(X):-
    write(X),
    eingabe(Y),
    einlesen(Y).

eingabe(X):-
    einlesen(X),
    writedevice(screen),
    write(X),
    writedevice(xausgabe).
```

2.
```
domains
    datei = xeingabe
```

```
predicates
    los
    einlesen

clauses
    los:-
    openread(xeingabe,"aus.dat"),
    readdevice(xeingabe),
    einlesen,
    closefile(xeingabe),
    readdevice(keyboard).

    einlesen:-
        eof(xeingabe),
        !.

    einlesen:-
        eingabe(Antwort),
        write(Antwort),
        einlesen.
```

Kapitel 17

2. Das Prädikat *not* kann keine Variable binden. Wenn Sie es versuchen, erhalten Sie die Fehlermeldung 3004.

Kapitel 18

3. Vorteile: In Prolog ist bereits der größte Teil des Inferenz-Mechanismus eingebaut. Dadurch beschränkt sich die Programmierarbeit im wesentlichen auf die Errichtung der Wissensbasis. Die Zeit für eine Programmentwicklung ist relativ kurz, und das Programm ist leicht zu lesen und von Fehlern zu befreien. Prolog unterstützt sehr komplexe Strukturen, so wie zum Beispiel zusammengesetzte Objekte und Listen (ebenso wie hierarchische Strukturen, die diese Komponenten enthalten).

Nachteile: Die Anwendung von Prolog ist im wesentlichen auf das Tiefensuchverfahren mit Rückwärtsverkettung und auf Produktionssysteme beschränkt. Man kann es zwar auch mit Vorwärtsverkettung und für rahmenbasierende Systeme einsetzen, aber das erfordert einen großen Programmieraufwand. Außerdem sind die numerischen Verarbeitungsmöglichkeiten in Prolog sehr begrenzt.

Kapitel 19

1. Um das Programm zu testen, gibt man jede Symptom-Gruppe der Original-Faktenliste ein. Dann muß jedesmal die entsprechende Krankheit als Ergebnis angezeigt werden und keine andere.

   Fügen Sie ein zweites Prädikat *diagnose("Gehirnhautentzündung")* hinzu:

```
diagnose("Gehirnhautentzündung'":-
     symptom(fieber),
     symptom(erbrechen),
     symptom(kopfschmerzen),
     symptom(hautausschlag),
     symptom(licht),
thema("Gehirnhautentzündung").
diagnose("Gehirnhautentzündung"):-
     symptom(fieber),
     symptom(erbrechen),
     symptom(kopfschmerzen),
     symptom(licht),
thema("Gehirnhautentzündung").
```

Anhang G

Turbo Prolog und andere Prolog-Versionen

Für die Sprache Prolog gibt es bis jetzt noch keinen Standard, wenn man einmal von dem Prolog absieht, wie es in dem Buch *Programming in Prolog* von Clocksin und Mellish (siehe Anhang E) beschrieben wird, das auch allgemein als C&M-Prolog bekannt ist. Die meisten kommerziellen Prolog-Versionen bilden eine Obermenge zu diesem Sprachfundament.

Turbo Prolog stellt sowohl eine Obermenge als auch eine Untermenge von C&M-Prolog dar. Es enthält viele Funktionen, über die das C&M-Prolog nicht verfügt, aber einige fehlen ihm auch. Zusätzlich unterstützt Prolog einige Funktionen durch Strukturen (wie z. B. Ein- und Ausgabe), die mehr an die Sprache C erinnern als an C&M-Prolog. Programme, die unter C&M-Prolog geschrieben wurden, laufen unter Turbo Prolog nur mit Änderungen, zumeist mit sehr großen Änderungen. Umgekehrt müssen auch Turbo Prolog-Programme stark verändert werden, bevor sie unter C&M-Prolog laufen können.

In dem Sinne ist Turbo Prolog eher eine neue Sprache, zwar mit vielen Ähnlichkeiten zu C&M-Prolog und einigen Ähnlichkeiten zu C. Beim Vergleich von Turbo Prolog mit C&M-Prolog können wir folgende Punkte festhalten:

- Turbo Prolog ist wesentlich schneller als alle C&M-Implementationen. Wir haben es mit einem echten Compiler zu tun.

- Die Notwendigkeit in Turbo Prolog, Typen zu deklarieren, ermöglicht dem Anwender eine bessere Kontrolle über die Compilierungs- und Ausführungsumgebung. Bei Sprachen ohne Typdeklarationen (wie z. B. FORTH) kann das Betriebssystem wegen Fehlern abstürzen, können Daten auf der Festplatte verlorengehen oder andere Probleme auftreten.

- Die aktuelle Turbo Prolog-Version unterstützt keine virtuelle Speicherverwaltung, wodurch die Größe der Regelbasis auf den verfügbaren Speicherplatz im Rechner beschränkt ist.

- Turbo Prolog bietet eine hervorragende Anwender-Schnittstelle mit Fenstern, zwei Arten von Grafik und Sound.

- Die Entwicklung eines Programms in Turbo Prolog geht sehr schnell. Der eingebaute Editor ist einfach anzuwenden, und der Compilier-Vorgang ist ebenfalls sehr schnell.

- Turbo Prolog-Programme laufen nicht unter anderen Prolog-Versionen.

- Turbo Prolog unterstützt mit eingebauten Prädikaten direkten Zugriff auf DOS und die Systemfunktionen.

- Turbo Prolog bietet logische Bit-Operatoren, umfangreiche Dateiverarbeitungsmöglichkeiten und leistungsfähige Ein/-Ausgabeoperationen.

- Die aktuelle Version von Turbo Prolog enthält nur eine Metasprachfunktion, das Prädikat *findall*. Alle anderen Metasprachfunktionen des C&M-Prologs sind nicht vorhanden. Diese Funktionen ermöglichen es einem C&M-Programm, sich selbst zu überprüfen, Programmcode wie Daten zu behandeln und komplexe Klauseln und Ziele zu errichten. Demzufolge sind fortgeschrittene symbolische Operationen in Turbo Prolog nur schwer zu realisieren.

- Die Unifikationsmöglichkeiten in Turbo Prolog sind beschränkter als in C&M-Prolog. Dort würde beispielsweise *Summe(X + Y)* mit *Summe(2 + 3)* unifizieren, indem X an 2 und Y an 3 gebunden wird. Auch der Ausdruck *prüf(motor,200)* wird dort mit *prüf(motor,Volt < 220)* unifizieren, indem Volt an 200 gebunden wird. In Turbo Prolog sind solche Ausdrücke nicht zugelassen.

Stichwortverzeichnis

Algorithmus 19
Arithmetik
  Festkomma 28
  Fließkomma 28
Ausgabe 89
  auf den Drucker 93
  bildschirmbezogene 190
  Umlenken 202

Backtracking 71 f.
BASIC 19
Beispielsprogramms
  Erstellen eines 34
Blockbefehle 37
Blockoperationen 40
Bündeln 248

C 19
COBOL 19
Compile 33
Compiler 27 f.
Compilierung 219
Compilierungsprozeß 34
Copy 46
Cursorsteuerung 28, 36 ff.

Dateien
  Direktzugriff auf 28
  Öffnen 203
  Schließen 203
Datenbank
  dynamische 153 ff.
  Speichern der 163
  statische 154 f.
  Verwendung der 153, 161
dBASE III 19
Determinismus 129 f., 223
Diagnose-Programm
  medizinische 153
Dialog 32
Directory 46
DOS
  Dateioperationen in 210

Edit 33
Editor 32
  verlassen 43
Editierbefehle 37
Eingaben 89
  bildschirmbezogene 190
  Umlenken 202
Erase 46
Ersetzen 42
Expertensysteme 24 f., 231 f.
  Aufbau 241
  Aufgaben des 231 f.
  Typen von 234

Fakten
  Beschaffung von 242
  Löschen von 162
  Tabellieren der 245
Farb-Monitor 187
Felder
  Ein- und Ausgabe 191
Fenster 32, 185
  Ein- und Ausgabe 192
  erstellen von 185
  Technik 189
  Verwendung von 99 f.
Fensterparameter 48
Fenstertechnik 28
Festplatte
  Arbeiten mit einer 279
Files 33

FORTRAN 18
Frequenzwerte 199
Funktionen
mathematische 137
verschiedene 42

Gleichheitszeichen 132 f.
Goal 20
Grafik 28, 185, 193

Hardware 275
Hilfsdateien
Verwendung von 209
Hilfsinformationen 43
Hintergrund
ändern 50

Interpreter 27

Kommandozeilen-Parameter
Verwendung von 213
Kopie
Anfertigen von 277
Erstellen einer 279

Laden 43
Linien 194
LISP 18
Listen 167 f.
arbeiten mit 167
Deklaration von 168
Domain-Typen in 175
Listen von 171
Operationen 173
Programmieren von 171
Schreiben von 173
Umkehren einer 174
Unifikation bei 168
Verketten von 173

Meldung
Copyright 31
Menü
File 33
Menüstrukturen
Erstellen von 164
Menüsystem
einfaches 165
Message 32
Monochrom-Monitor 186

Objekte
Ausschluß von 107 f.
Informationen zu 148
Unterscheiden von 149
Operatoren
arithmetische 131, 138
Options 33

Pascal 19
Prädikat 86 f.
logische 139
mathematisch 139
Verwendung
von arithmetischen 140
Print 46
Produktionssysteme 234
Programm
ausführen 47
compilieren 47
editieren 36
Entwurf 215
modulares 225
steuern 84
schreiben 248
Stil 215
Techniken 216
Test 259
Tricks 216
Programmablaufverfolgung 220
Prolog
Anwendung in 24 f.
Bibliothek 225
Einführung in 17
Entwicklung von 18 f.
Leistungsfähigkeiten 21
Variablen in 65

Puffer
Entleeren der 207
Punkte 194

Quit 33

Rahmenattribut 188
Regeln 75
Rekursion 109 f.
Grundregeln der 114
Rename 46
Robotik 26
Run 33

Schlußfolgerungs-Automat 236
Schnitt 119 ff.
Einsatzschwerpunkt 121
Setup 33
Software 277
Sound 28, 185, 193, 198
Speichern 43
Spielentwurf 263
Sprachen
Objekt-orientierte 21
prozedurale 21 f., 28
Standard-Operatoren
arithmetische 137
Standard-Prädikate 85 f.
Verzeichnis der 283
Suchen 42
Systeme
rahmenbasierende 238

Text
einfügen 39
löschen 39
Trace 32
Turbo Prolog
arbeiten mit 31
Aufbau eines 227
Datenverarbeitung 201
Eigenschaften von 28
Einschränkung von 28
Installierung von 275
Objekte in 53
Programmentwicklung in 215
Starten von 31 f.
Zeichenketten in 177 f.
Turtle-Grafik 197
Typumwandlung 143

Überlaufkontrolle 222
Unifikation 82 f.
Unterbrechungen 222

Variablen 77 f.
anonyme 68
freie 67 f.
gebundene 67 f.
Vereinbarung für 72
Verwendung von 66
Variablenwerte
Testen von 140
Vergleichsoperatoren 134
Verketten 178

Warnungen 222
Wissensbasis 235
Wissensdarstellung 233
WordStar 36
Wörter
reservierte 293

Zähler 141
Zeichen
Ein- und Ausgabe 190
Zeichenkettenoperationen 177 f.
Zeichenkettenverarbeitung 28
Zeichenstift
Steuerung 197
Zufallszahlen
Erzeugen von 143
Zusammengesetzte Operatoren 145
Deklaration von 149
Eingabe von Daten in 150
Gebrauch von 147
Verwendung von 145

Die SYBEX-Bibliothek

IBM PC

DAS HANDBUCH FÜR IBM PC UND KOMPATIBLE
von Horst Bodemann – Das Buch für Besitzer IBM-Kompatibler Rechner, die eine zusammenfassende Information über ihr System, das Betriebssystem MS-DOS, EDLIN und die Arbeit mit GW-BASIC suchen. 368 Seiten, 56 Abb., Best.-Nr. **3633** (1986)

IBM PC UND KOMPATIBLE – DIE TURBO DATABASE TOOLBOX
von Matthias Grafen – Anhand praxisorientierter Beispiele wird dem Leser der Umgang mit dem leistungsfähigen Dienstprogramm für die Programmentwicklung unter Turbo Pascal vertraut: Arbeiten mit Turbo Access, Arbeiten mit Turbo Sort, Fehleranalyse, Übungen und eine Fülle notwendigen Hintergrund-Wissens. 240 Seiten, mit Abb., Best.-Nr. **3659** (1987)

STARKONTOR PC: TEXTVERARBEITUNG
von Reinhold Krumscheid – Das bedienerfreundliche Textverarbeitungsprogramm für IBM PC und Kompatible – auch für weniger geübte Anwender leicht zu nutzen. Durch die einfache Funktionswahl aus dem Hauptmenü (über Cursortasten oder durch direkte Eingabe des Anfangsbuchstabens) ist ein in hohem Maße ergonomisches Editieren, Formatieren und Umbrechen von Texten möglich. Mit direkter Anzeige von Textattributen auf dem Bildschirm und Installationsmodul für IBM- oder EPSON-kompatible Drucker. Diskette + Trainingsbuch, Best.-Nr. **4004** (1987)

STARKONTOR PC: DATEIVERWALTUNG
von Stephan A. Horvath/Markus Weidemann – Das leistungsfähige und komfortable Dateiverwaltungs-Programm für die Nutzer eines IBM PC oder kompatiblen Rechners aus dem StarKontor-Paket. Diskette + Trainingsbuch, Best.-Nr. **4005** (1987)

STARKONTOR PC: ADRESSVERWALTUNG
von Stephan A. Horvath/Markus Weidemann – Das bedienerfreundliche Adreßverwaltungsprogramm für IBM PC und Kompatible: Jede Adresse mit frei editierbarer Infodatei; bequemes Suchen nach unterschiedlichen Kriterien sowie logischen Verknüpfungen; leichtes Korrigieren und Editieren bestehender Felder; schnittstellenkompatibel zum Fakturier- wie auch zum Textverarbeitungsprogramm der StarKontor-Serie. Diskette + Handbuch, Best.-Nr. **4010** (1986)

STARKONTOR PC: FAKTURIERUNG
von Stephan A. Horvath/Markus Weidemann –Nutzer eines IBM PC oder kompatiblen Rechners lösen ihre Fakturierungsaufgaben im Betrieb effizienter mit diesem komfortablen und bedienerfreundlichen Programm; schnittstellenkompatibel zur StarKontor-Adreßverwaltung sowie Artikel- und Lagerverwaltung. Diskette + Handbuch, Best.-Nr. **4011** (1986)

STARKONTOR PC: FINANZBUCHHALTUNG
von Stephan A. Horvath/Markus Weidemann – Mit dem leistungsfähigen und sehr bedienerfreundlichen Programm aus der StarKontor PC-Serie wickeln Anwender im Betrieb ihre Finanzbuchhaltung problemlos auf dem eigenen IBM PC oder kompatiblen Rechner ab. Mit frei definierbarem Kontenrahmen (300 Konten). Diskette + Trainingsbuch, Best.-Nr. **4012** (1987)

STARKONTOR PC: ARTIKEL- UND LAGERVERWALTUNG
von Stephan A. Horvath/Markus Weidemann – Ein weiterer Schwerpunkt der betrieblichen Verwaltungsarbeit, die Artikel- und Lagerverwaltung, läßt sich mit dem entsprechenden Programm der StarKontor PC-Reihe einfacher und effektiver gestalten. Auf einem IBM PC oder Kompatiblen und mit dem gewohnten StarKontor-Komfort. Mit Schnittstelle zum StarKontor Fakturierprogramm. Diskette + Trainingsbuch, Best.-Nr. **4013**(1986)

STARKONTOR PC: LOHN UND GEHALT
von Stephan A. Horvath/Markus Weidemann – Mit dem Lohn- und Gehaltsprogramm wird die StarKontor-Serie richtig rund. Nutzer eines IBM PC oder Kompatiblen machen sich damit unabhängig von externen Beratern. Wegen der hohen Bedienerfreundlichkeit auch ohne profunde Vorkenntnisse direkt nutzbar. Diskette + Trainingsbuch, Best.-Nr. **4014** (1987)

STARKONTOR PC-DOS MANAGER
von Stephan A. Horvath/Markus Weidemann erleichtert allen Anwendern den Umgang mit ihrem PC, die den umfangreichen und teilweise nicht unkomplizierten Befehlssatz parat haben wollen. Die wichtigsten Operationen – wie Kopieren von Dateien oder Disketten, Formatieren, Abruf von Inhaltsverzeichnissen u.v.m. – werden menügesteuert ausgewählt und aktiviert. Diskette + Trainingsbuch, Best.-Nr. **4015** (1986)

Systemsoftware

MS-DOS HANDBUCH
von Richard Allen King – Überarbeitete Ausgabe mit Version 3.2. Das unentbehrliche Standardwerk für Anwender des Betriebssystems MS-DOS, abgerundet durch Tabellen, Zeichnungen und viele praktische Beispiele, hilft Ihnen, das Beste aus Ihrem System herauszuholen. 384 Seiten, 44 Abb., Best.-Nr. **3644** (1987)

SYBEX RATGEBER MS-DOS
von Wolfgang Höfs – Die sichere Information mit kurzem Zugriff für PC-Anwender, die ihr Betriebssystem voll ausnutzen wollen. Mit Nachschlageteil, Fallbeispielen, Piktogrammen und Querverweisen. 450 Seiten, Best.-Nr.: **3302** (1986)

Kommunikation

DAS MODEMBUCH ZUR DFÜ
von Bruno Hurth und Manfred Hurth – Jetzt steht Ihnen ein Nachschlagewerk mit einer Fülle unentbehrlicher Informationen zur Verfügung, auf das Sie immer wieder zurückgreifen werden: Möglichkeiten der DFÜ im Bereich der DBP, Mailboxen, Akustikkoppler, Btx u.v.m. 224 Seiten, Best.-Nr.: **3619** (1985)

SYBEX MAILBOX FÜHRER
Alles über den Zugang zu elektronischen Briefkästen in Deutschland: Voraussetzungen für die Teilnahme an der Datenkommunikation; Telefon-Nummern, Eingangs- und Untermenüs wichtiger Mailboxen und was diese Ihnen nutzen können, u.v.m. 272 Seiten, Best.-Nr.: **3663** (überarbeitete Ausgabe 1986)

ONLINE DATENBANKEN
Zugang zum Wissen der Welt mit Mikrocomputern von **Steffen Schubert** – Manager aller Unternehmensgruppen, Wissenschaftler und Ingenieure erhalten einen Überblick über alle Datenbank-Typen, bekommen wichtige Anbieter in Europa und Übersee genannt und lernen, Datenbanken effektiv und kostengünstig zu nutzen. 200 Seiten, Best.-Nr. **3621** (1986)

Anwendungssoftware

LOTUS 1-2-3. DATENVERARBEITUNG OHNE VORKENNTNISSE
von S. Heine – für alle, die ohne DV-Kenntnisse das starke Software-Paket LOTUS 1-2-3 für berufliche oder private Anwendungen einsetzen möchten. 264 Seiten, 64 Abb., Best.-Nr. **3052** (1985)

ARBEITEN MIT LOTUS 1-2-3
von B. F. Kehlmann – die wichtigsten Anwendungsfunktionen von LOTUS 1-2-3 im Betrieb anhand praktischer Fallstudien. Überarbeitete und erweiterte Ausgabe für die deutsche LOTUS-Version, ca. 250 Seiten, mit Abb., Best.-Nr. **3643** (1987)

ARBEITEN MIT SYMPHONY
von Douglas Cobb – Ein umfassendes Standardwerk für professionelle Anwender mit der Version 1.2. Für den Kenner von Symphony werden alle Erweiterungen von Symphony 1.2 deutlich und schnell erkennbar herausgearbeitet. 872 Seiten, 420 Abb., Best.-Nr. **3095**(1986)

ARBEITEN MIT WORDPERFECT
von Susan Baake Kelly – WordPerfect setzte sich als leistungsfähiges Textverarbeitungs-Programm schnell am Markt durch und erfreut sich auch hierzulande wachsender Beliebtheit. Kompetent und Schritt für Schritt wird der Leser in die Arbeitsumgebung des Softwarepaketes eingeführt. Die Autorin hat das Arbeitsbuch so angelegt, daß sowohl Starter wie auch erfahrene Anwender daraus ihren Nutzen ziehen können. Das Werk ergänzt in hervorragender Weise das Handbuch, wobei auch spezielle Themen detailliert erläutert werden: Datensicherung, Verknüpfen von Texten, Sortier- und Auswahl-Operationen, Rechnen im Text, Benutzen von Macros und der Umgang mit dem Rechtschreib-Korrekturprogramm. Informative Anhänge machen das Arbeitsbuch komplett. Ca. 450 Seiten, mit Abb., Best.-Nr. **3690** (1987)

SYBEX RATGEBER dBASE III
von Gerhard Renner – Das Nachschlagewerk zu dBase III für alle, die sich mit ihrer Software auskennen, aber ganz schnell Detail-Informationen benötigen. Übersichtlicher Nachschlageteil mit Piktogrammen und Querverweisen. 432 Seiten, Best.-Nr. **3300** (1986)

ARBEITEN MIT dBASE III
von Alan Simpson – zeigt Ihnen, wie Sie Ihre Datenbank-Verwaltung mit dBASE III noch effektiver machen; mit vielen nützlichen Beispielen. 288 Seiten, mit Abbildungen, Best.-Nr. **3609** (1986)

SYBEX TRAINER MS WORD
von Dr. G. Renner – Der Textverarbeitungskurs, der Ihnen einen schnellen Einstieg in MS-Word gibt. In Theorie und Praxis zugleich durch die Wechselbeziehung zwischen einführendem Text und direkt gegenüber stehenden Bildschirm-Abbildungen. 344 Seiten, 175 Abb., Best.-Nr. **3330** (1986)

ARBEITEN MIT dBASE III PLUS
von Alan Simpson/Gerhard Renner – führt den Leser in die Arbeitsumgebung des Programms ein und erläutert anhand einer Vielzahl aussagekräftiger Beispiele die Stärken des Software-Paketes. Für dBASE III plus-Anwender, Umsteiger von dBASE III und Interessenten im Vorfeld der Kaufentscheidung. 408 Seiten, ca. 140 Abb., Best.-Nr. **3635** (1987)

Andere Programmiersprachen

ERFOLGREICH PROGRAMMIEREN MIT C
von J. A. Illik – ein unentbehrliches Handbuch für jeden, der mit der universellen Sprache C erfolgreich programmieren will. Aussagekräftige Beispiele, auf verschiedenen Mini- und Mikrocomputern getestet. 408 Seiten, Best.-Nr.: **3055** (1984)

C – EINE EINFÜHRUNG
von Bruce H. Hunter – Das ideale Buch für den Einsteiger in die Programmiersprache C, speziell für Anwender, die von BASIC auf den leistungsfähigen Compiler umsteigen wollen. 296 Seiten, ca. 12 Abb., Best.-Nr. **3632** (1986)

Pascal

DAS TURBO PASCAL BUCH
von Karl-Hermann Rollke – Sie lernen die Arbeitsweise des Turbo-Editors und des Systems kennen und werden mit Programmierkonzepten, Daten- und Kontrollstrukturen für den Programmfluß vertraut gemacht. 288 Seiten, mit Abbildungen, Best.-Nr.: **3608** (1985)

EINFÜHRUNG IN PASCAL MIT TURBO PASCAL
von Rodnay Zaks – Der Sybex-Longseller liegt jetzt in einer an Turbo Pascal angeglichenen Version vor, die den Nutzen für an der Sprache Pascal Interessierte nochmals wesentlich erhöhen wird. 464 Seiten, ca. 140 Abb., Best.-Nr. **3645** (1986)

SYBEX

Fordern Sie ein Gesamtverzeichnis unserer Verlagsproduktion an:

SYBEX-VERLAG GmbH
Vogelsanger Weg 111
4000 Düsseldorf 30
Tel.: (02 11) 61 80 2-0
Telex: 8 588 163

SYBEX INC.
2021 Challenger drive, NBR 100
Alameda, CA 94501, USA
Tel.: (4 15) 523-8233
Telex: 287 639 SYBEX UR

SYBEX
6–8, Impasse du Curé
75018 Paris
Tel.: 1/203-95-95
Telex: 211.801 f

SYBEX

Einführung in Turbo Prolog

Das Buch führt den interessierten Leser in die Programmiersprache Turbo Prolog und damit in den Themenbereich der künstlichen Intelligenz ein.

Anhand von Beispielen werden Wege zur Entwicklung von Expertensystemen vorgestellt, die die Leistungsfähigkeit von Prolog anschaulich unter Beweis stellen.

Aus dem Inhalt:

- Einführung in Prolog
- Arbeiten mit Prolog
- Fakten, Objekte und Prädikate
- Regeln
- Programmablaufsteuerung
- Arithmetische Operationen
- Dynamische Datenbanken
- Fenster, Grafik, Sound
- Einführung in Expertensysteme

Ein Anhang mit wichtigen Informationen ergänzt das kompetent geschriebene Werk.

ISB N 3-88745-680-7

DM 48,—
sFr 44,20
öS 374,—

9 783887 456801